High Performance Traffic Monitoring for Network Security and Management

Tristan Groleat

N° d'ordre : 2014telb0316

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En accréditation conjointe avec l'école Doctorale Sicma

High performance traffic monitoring for network security and management

---

Thèse de Doctorat

Mention : STIC

Présentée par **Tristan Groléat**

Départements : Informatique, électronique
Laboratoires : IRISA REOP, LabSTICC CACS

Directrice de thèse : **Sandrine Vaton**

Encadrant : **Matthieu Arzel**

Soutenue le 18 mars 2014

Jury :

M. Guy Gogniat, Professeur, Université de Bretagne Sud (Président)
M. Philippe Owezarski, Chargé de Recherches, LAAS/CNRS (Rapporteur)
M. Dario Rossi, Professeur, Télécom ParisTech (Rapporteur)
Mme Sandrine Vaton, Professeur, Télécom Bretagne (Directrice de thèse)
M. Matthieu Arzel, Maître de Conférences, Télécom Bretagne (Encadrant)
Mme Isabelle Chrisment, Professeur, Télécom Nancy
M. Stefano Giordano, Professeur, University of Pisa
M. Ludovic Noirie, Chercheur Senior, Alcatel Lucent

Remerciements

Je remercie d'abord Sandrine Vaton, ma directrice de thèse, qui a été très présente tout au long de ces trois ans et trois mois de thèse. Elle m'a conseillé, m'a fait profiter de son expertise sur les algorithmes de surveillance réseau, et m'a aidé à valoriser les résultats obtenus. Je remercie aussi Matthieu Arzel qui m'a encadré, et m'a fait profiter de son expertise en électronique numérique. Je remercie Sandrine et Matthieu pour le temps qu'ils ont passé à relire et commenter chacun de mes articles, chacune de mes présentations, ainsi que ce manuscrit. Mais je les remercie surtout pour m'avoir permis de découvrir la possibilité de lier la surveillance de trafic et l'électronique, en proposant un projet de détection d'attaques alors que j'étais encore élève ingénieur à Télécom Bretagne. Ils m'ont ensuite soutenu quand j'ai souhaité prolonger ce travail par une thèse, en m'aidant à transformer une simple idée en véritable sujet de thèse, et en trouvant des financements grâce à PRACOM et au projet européen DEMONS.

En parallèle de la recherche, Télécom Bretagne m'a aussi donné la possibilité de participer aux enseignements, ce qui fut très enrichissant. Je remercie Sylvie Kerouédan, qui a supervisé ma mission d'enseignement. Elle a aussi créé durant ma thèse le Téléfab, le FabLab de Télécom Bretagne, lieu permettant à tous d'échanger et de créer des objets de tous types simplement à partir d'une idée, ce qui m'a permis de faire de nombreuses rencontres et découvertes, en tant qu'étudiant et en tant qu'encadrant.

Merci à Olivier Emery, Hicham Bougdal, Sébastien Martinez, Alban Bourge, Yannick Le Balch et Manuel Aranaz Padron pour avoir fait avancer mon sujet de thèse durant un stage ou projet.

Je remercie aussi toutes les personnes à Télécom Bretagne qui ont facilité mon travail de thèse, et particulièrement Patrick Adde qui a fait le lien avec PRACOM, et Armelle Lannuzel et Catherine Blondé, qui ont du gérer mon inscription dans deux départements à la fois. Merci à Bernard L'hostis qui a du souvent me trouver des PCs, de la RAM ou des écrans.

Je remercie toutes les personnes au département électronique qui ont rendu le travail quotidien très agréable, et en particulier Benoît Larras, Pierre-Henri Horrein, Kévin Burgi, Gérald Le Mestre, Jean-Noël Bazin, Valentin Mena Morales, Charbel Abdel Nour et Michel Jezequel.

Je remercie enfin ma famille et mes proches qui m'ont toujours soutenu, même à distance.

Contents

Chapter A

Abstract

Traffic monitoring is a mandatory task for network managers, be it small company networks or national Internet providers' networks. It is the only way to know what is happening on a network. It can be very basic and consist in simply measuring the data rate of the traffic on a link, or it can be more sophisticated, like an application to detect attacks on protected servers and raise alarms.

Use cases for traffic monitoring are very diverse. They can be separated into categories. An important category is security: a large number of attacks are transmitted using the networks of Internet providers everyday. These attacks are dangerous, and some of them can saturate network resources and degrade the quality of service offered by the network. So network managers are very interested in detecting these attacks to try and mitigate them. Another important category is traffic engineering. It is important for all network managers to get as much data as possible about what happens on their network. Data can be used simply for statistics, or it can be used in real time by automated systems: knowing what the traffic is made of is a good way to make better real time network management decisions.

As Internet keeps gaining importance in our societies, the exchanged data amounts increase every year. To support these increases, network operators install new links with very high data rates. Customers can now get a connection of 1 Gb/s by optical fiber, so aggregation links of tens of gigabits per second are common. This is why traffic monitoring applications now have to support high data rates to be deployable on real networks.

To implement traffic monitoring applications that support high data rates, we first analyze four existing development platforms: powerful computers with high-speed network interfaces, powerful computers with high-speed network interfaces and a fast graphic card to accelerate processing, network processors and Field-Programmable Gate Arrays (FPGAs). Network processors are like normal processors, but with an optimized access to network interfaces and hardware-accelerated features useful for traffic processing. FPGAs are chips that are configurable at a very low level. They offer massive parallelism, a very low-level access to network interfaces, and a perfect control of timing.

Normal computers are the most flexible and cheapest option, but they struggle handling received packets at high data rates. Graphic cards can accelerate highly

parallel computations but cannot help for low level packets processing. Network processors are good at packet processing and can offer good performance, but they are specialized for some specific use cases, different for each model. FPGAs support high data rates and can accelerate processing using parallelization, but the development time is long and the resulting application is difficult to make flexible.

Due to these differences, no platform is the absolute best, it depends on the use case. This is why we study three different use cases. All require the support of high data rates but the main goals are different. The first use case is in the domain of security and focuses on flexibility. It allows to explore the possibilities to support high data rates in software when implementing light algorithms, while keeping the great flexibility of software development. In contrast, the second use case in the domain of traffic engineering requires the implementation of heavy algorithms in a fully real-time environment. This is a reason to explore the possibilities offered by FPGAs to accelerate complex processing. It shows how parallelism and low-level control can help manage high data rates. The last use case combines the requirements of the two previous ones. It is a traffic generator with strong real-time needs, but for which flexibility is essential too. The chosen solution is a trade-off using an FPGA in collaboration with a processor to configure it.

The first use case in the security domain was done in the framework of a European project called DEcentralized, cooperative, and privacy-preserving MONitoring for trustworthinesS (DEMONS). We participated to the development of a flexible and scalable network monitoring framework for normal computers. It is a pure software development. We developed an application to test the framework: a detector of a certain kind of network attacks called Transmission Control Protocol (TCP) SYN flooding. We used an algorithm for counting packets called Count Min Sketch (CMS) and an algorithm for detecting abrupt changes in time series called CUmulative SUM control chart (CUSUM). The advantage of software development is that it is very flexible: the detector can be made distributed simply by changing the configuration using a Graphical User Interface (GUI). We show that thanks to the efficient framework, the application is able to support realistic 10 Gb/s traffic, but it struggles in stress conditions when receiving very small packets. We then give ideas about the possibility to build an hybrid architecture, with an FPGA managing the packets and sending aggregated data to the computer. The goal is to keep the flexibility of software while supporting higher data rates easily.

After showing the limits of pure software development, we turn to pure hardware development on FPGA with the use case of traffic classification. It consists in associating each packet transiting on a link to the category of application that generated it: for instance a web browsing application, a video streaming application, or a gaming application. We use a well-known classification algorithm called Support Vector Machine (SVM). It works with a learning phase and a classification phase. The classification phase is implemented using an optimized, massively parallel architecture on the FPGA. We show that a unique FPGA could handle the computation phase of the classification of a realistic trace at a rate up

to 473 Gb/s if adapted interfaces were available. To let the application work, we also need a way to group received packets into flows of similar packets, supporting at least one million simultaneous flows. We develop a new algorithm to do that and implement it on the FPGA. An actual implementation on an existing board supports the link rate (10 Gb/s) without problems. The main drawback of the implementation is the flexibility: changing parameters of the algorithm forces to reconfigure the FPGA, which takes time.

To test the attack detector or the traffic classifier, a traffic generator is needed. As test implementations support 10 Gb/s, the same rate has to be supported by the traffic generator. And to get reliable results, the generator should be able to generate traffic accurately at any rate up to 10 Gb/s, even when sending small packets, which are the most challenging to handle. As commercial traffic generators are expensive and not very flexible, we build our own open-source flexible and extensible traffic generator supporting a data rate up to 20 Gb/s. To build a reliable generator, we decide to implement it on an FPGA. But for this application, flexibility is required. All parameters of the generated traffic must be easy to configure: data rate, packet size, inter-packet delay, packet headers and data. So we design a very flexible modular architecture for the traffic generator. Each module can be configured without FPGA reconfiguration, and the creation of new modules modifying the generated traffic is made as easy as possible. A GUI is built to make the configuration simple. We show that the traffic generator can support up to 20 Gb/s easily, and that it respects the configured data rate with a very good accuracy.

Many high-performance tools built for our test implementations can be reused for other traffic monitoring applications: counting packets using CMS (software), detecting changes in time series using CUSUM (software), classifying data using SVM (hardware), grouping packets into flow using a custom algorithm (hardware), generating traffic in a flexible and scalable way (hardware). These different experiments on commodity hardware and on FPGA also allow us to draw conclusions on the best platform to use depending on the use case.

Chapitre B

Résumé

B.1 Introduction

La surveillance de trafic est une activité indispensable pour tous les gestionnaires de réseaux, que ce soit des réseaux de petites entreprises ou de fournisseurs nationaux d'accès à internet. C'est le seul moyen pour savoir ce qui se passe sur le réseau. L'objectif peut être très simple, comme la mesure du débit d'un lien, ou plus sophistiqué, comme la détection d'attaques contre des machines protégées.

Un aspect important de la surveillance de trafic est la sécurité. Les réseaux sont des outils pour différents comportements illégitimes : propagation de virus, envoi de courriers non sollicités, recherche de machines vulnérables, prise de contrôle de machines sans autorisation, envoi de grandes quantités de données pour surcharger un serveur, ou interception de données sensibles. La plupart de ces comportements produit du trafic spécifique qui peut être identifié par des outils de surveillance de trafic. Mais faire la différence entre un trafic légitime et illégitime n'est pas évident, surtout si les attaquants tentent de faire passer leur trafic pour du trafic normal. Récemment, de nombreuses attaques de grande envergure ont eu lieu, comme par exemple une tentative de faire tomber l'infrastructure du NASDAQ [Rya13]. Chaque machine peut être protégée contre les attaques par des mises à jour régulières pour colmater les failles découvertes, et par l'utilisation d'un antivirus et d'un pare-feu. Mais les opérateurs ont également leur rôle à jouer. En effet, ils ont une vision globale du réseau, et ils sont les seuls à avoir la possibilité de stopper une attaque à sa source. Actuellement, les opérateurs laissent leurs clients se défendre, mais les plus grosses attaques peuvent mettre à mal leur réseau. Ils ont donc intérêt à offrir des services de protection avancés à leurs clients. Pour cela, ils doivent disposer d'outils de surveillance en temps réel de leur réseau.

Un autre aspect important de la surveillance de trafic est son utilisation pour l'ingénierie de trafic. L'objectif est de mieux connaitre le trafic pour mieux le transporter. Un application importante pour cela est la classification de trafic. Le trafic sur Internet est fait d'une agrégation de paquets représentant des bouts d'information. Chaque paquet contient une en-tête qui indique où envoyer le paquet et d'où il vient, mais pas quelle application l'a généré ou quel type de trafic il transporte. La classification de trafic consiste à associer un paquet à

l'application qui l'a généré. De cette manière, un paquet généré par Skype peut être traité différemment d'un paquet généré par Firefox par exemple. En effet, les besoins en termes de délai d'une communication vocale et de la consultation d'une page web sont très différents. La classification de trafic peut aussi être utile pour filtrer un type de trafic à surveiller particulièrement.

Bien que les applications de surveillance de trafic puissent être très différentes, elles font toutes face à une difficulté commune : la montée en débit. CISCO prévoit que le trafic global sur Internet va tripler entre 2012 et 2017 [Cis13]. La surveillance de trafic doit donc se faire à très haut débit. Les liens de plusieurs dizaines de Gb/s deviennent fréquents chez les opérateurs. Ces débits rendent très difficile l'utilisation d'ordinateurs puissants du commerce pour surveiller le trafic, mais différentes plateformes plus adaptées existent. Des cartes réseau capables de gérer 2x10 Gb/s sont vendues par Intel. Pour la partie calculatoire, des cartes graphiques permettent d'accélérer les traitements. Pour des débits plus élevés, il existe des processeurs spécialisés pour les applications réseau. À encore plus bas niveau, les Field-Programmable Gate Arrays (FPGAs) sont des puces qui peuvent être configurées pour se comporter n'importe quel circuit numérique. Ces puces permettent un parallélisme massif et un accès direct aux interfaces réseau.

L'objectif de cette thèse est d'évaluer différents moyens d'accélérer des applications de surveillance de trafic. Une littérature abondante existe sur la détection d'anomalies et la classification de trafic. La précision des algorithmes étudiés est souvent mesurée avec soin, mais la possibilité de les réaliser de manière efficace est moins souvent analysée. C'est pourquoi nous allons d'abord étudier différentes plateformes de développement offrant des possibilités d'accélération logicielle ou matérielle. Nous allons ensuite réaliser une première application dans le domaine de la sécurité en utilisant l'accélération logicielle. Puis nous réaliserons un classificateur de trafic utilisant l'accélération matérielle. Enfin, nous trouverons un compromis entre logiciel et matériel pour réaliser un générateur de trafic flexible, fiable et rapide.

B.2 Choisir une plateforme de développement

Il existe plusieurs plateformes de développement adaptées aux applications réseau. Selon les besoins, le meilleur choix de plateforme peut être différent. Nous avons étudié trois plateformes. La première et la moins coûteuse est simplement un ordinateur du commerce puissant, avec une carte réseau capable de supporter des débits d'au moins 10 Gb/s. Un processeur graphique peut être ajouté à l'ordinateur pour permettre des calculs massivement parallèles. La seconde est un processeur réseau. C'est un processeur classique avec un accès direct à des interfaces réseau, et un jeu d'instructions qui permet de réaliser simplement certaines fonctions communes pour le traitement de trafic. La troisième est la plus bas niveau : les FPGAs. Ces puces peuvent être programmées pour réaliser n'importe quel circuit électronique. Elles permettent un accès direct à des interfaces réseau, et un traitement complètement parallélisé. Les avantages et inconvénients de chaque plateforme sont résumés en anglais dans les tableaux 2.1 et 2.2

Si le besoin le plus crucial pour l'application est la flexibilité, l'utilisation d'un

ordinateur du commerce semble être la meilleure solution car le développement et les mises à jour sont plus simples. Selon les besoins de parallélisme de calcul, l'utilisation d'un processeur graphique peut être nécessaire ou pas. Mais la vitesse de communication entre l'interface réseau et le processeur est très vite bloquante, même pour supporter seulement 10 Gb/s.

C'est pourquoi si l'objectif principal est d'obtenir le meilleur débit possible et la plus grande puissance de calcul, le choix se portera plutôt sur les processeurs réseau et les FPGAs. Les processeurs réseau sont très efficaces pour les tâches communes dans le domaine des réseaux (calcul de checksums...) et peuvent s'avérer peu coûteux s'ils sont vendus par centaines de milliers. En revanche, les FPGAs sont plus versatiles et le code développé est plus facile à porter vers d'autres plateformes. Si plus de puissance de calcul est requises et si le produit doit être vendu par centaines de milliers, les Application-Specific Integrated Circuits (ASICs) sont une option intéressante. Il s'agit de puces spécialement réalisées pour une tâche particulière. Le code développé sur FPGA peut être utilisé comme base pour concevoir un ASIC.

Dans les prochaines sections, trois applications vont être développées. Une sur un ordinateur du commerce, et les deux autres sur FPGA. Les résultats permettront de clarifier certains aspects de ces plateformes. En particulier :

- la comparaison de la puissance de calcul disponible sur processeur et sur FPGA, ainsi que la façon d'utiliser au mieux la puissance disponible ;
- la simplicité de mise à jour permise par les processeurs et les moyens d'obtenir cette simplicité sur FPGA aussi ;
- le temps de développement sur chaque plateforme, et les méthodes pour réduire ce temps pour les FPGAs ;
- les perspectives de futur passage à l'échelle permises par chaque plateforme.

B.3 Surveillance logicielle pour la sécurité

La façon la plus simple de réaliser une application de surveillance de trafic est de faire du développement logiciel sur un ordinateur du commerce. C'est la solution que nous avons étudié dans le cadre du projet européen DEcentralized, cooperative, and privacy-preserving MONitoring for trustworthinesS (DEMONS). L'un des objectifs de ce projet est de fournir une plateforme flexible de surveillance de trafic à haut débit. Notre rôle dans DEMONS est d'aider au développement et de tester cette plateforme appelée BlockMon. Pour cela, nous réalisons une application de détection d'un certain type d'attaques appelé Distributed Denial of Service (DDoS) qui consiste à envoyer depuis de nombreux ordinateurs beaucoup de trafic à un serveur pour l'empêcher de fonctionner normalement.

Le code développé pour BlockMon est en C++. Le fonctionnement de BlockMon est optimisé pour éviter autant que possible les opérations chronophages comme la copie des données en mémoire. BlockMon intègre une pile réseau optimisée appelée PFQ. Cette pile remplace la pile réseau standard de Linux et

est compatible avec la technologie Receive Side Scaling (RSS) d'Intel qui permet de séparer le trafic réseau reçu dans plusieurs files qui peuvent être traitées par des cœurs logiciels différents. BlockMon permet aussi de gérer très finement l'organisation du parallélisme grâce à la notion de groupes de processus réservés à l'exécution de certains blocs de code.

Grâce à sa structure en blocs configurables, BlockMon est très flexible et peut être configuré facilement grâce à une interface graphique. Sans aucun développement supplémentaire, une application développée correctement peut fonctionner avec une sonde unique ou avec des sondes multiples distribuées dans le réseau. Il est aussi possible d'utiliser plus ou moins de cœurs pour une application donnée, et de régler les paramètres de fonctionnement de l'application.

Le développement de l'application de détection de DDoS est simplifié par les blocs de base fournis par BlockMon. Nous avons développé deux algorithmes de base comme des librairies : Count Min Sketch (CMS) pour compter les paquets reçus avec une utilisation mémoire efficace, et CUmulative SUM control chart (CUSUM) pour détecter des changements brusques dans les nombres de paquets reçus. L'application elle-même est divisée en blocs pour exploiter la flexibilité offerte par BlockMon. Tous les blocs et librairies peuvent être réutilisés pour d'autres applications. Les librairies sont les plus simples à réutiliser car elles sont très génériques.

Pour tester le débit maximum supporté par notre application, nous avons utilisé un générateur de trafic pour envoyer un trafic difficile à notre détecteur. Le détecteur était installé sur une machine puissante avec une interface réseau à 10 Gb/s. Les résultats montrent l'intérêt de BlockMon et de son mécanisme de gestion du parallélisme : presque tous les paquets sont perdus en utilisant un seul cœur, mais moins d'un quart des paquets sont perdus avec sept cœurs. Ces résultats sont pour du trafic de test. Pour du trafic réaliste, avec des paquets plus longs, les 10 Gb/s peuvent être supportés par notre détecteur sans perdre de paquets.

Le développement logiciel est relativement simple et rapide. La flexibilité de BlockMon serait difficile à offrir avec des développements matériels, et l'exemple du détecteur de DDoS montre qu'il est possible de supporter 10 Gb/s en logiciel. Cependant, une machine très puissante est nécessaire, et les test avec le trafic le plus difficile à gérer montre que nous atteignons les limites de cette plateforme. En outre, une configuration très fine de la machine est nécessaire pour obtenir ces résultats.

Il serait intéressant de mélanger des fonctions bénéficiant d'accélération matérielle et des fonctions logicielles. Une société appelée Invea-Tech a déjà développé un block BlockMon permettant de communiquer avec un FPGA. Il serait donc possible par exemple d'utiliser une version matérielle de l'algorithme CMS, et d'exécuter seulement l'algorithme CUSUM en logiciel, car les contraintes temps-réel sont moins grandes pour CUSUM qui ne traite pas directement les paquets mais des données agrégées.

B.4 Surveillance matérielle pour la classification de trafic

Nous avons vu les possibilités et les limites des approches purement logicielles. Maintenant nous allons nous intéresser à une approche purement matérielle sur FPGA. Pour tester les capacités de calcul des FPGAs, nous nous intéressons à une application plus lourde en calcul que la détection de DDoS : la classification de trafic en fonction du type d'application (web, jeu, partage de fichiers...). Pour que l'algorithme soit capable de classifier aussi le trafic chiffré, nous utilisons des paramètres très simples pour classifier les paquets. Le trafic est divisé en flots, qui sont des ensembles de paquets ayant les mêmes adresses et ports source et destination. Un flot est caractérisé par la taille des paquets 3, 4 et 5. Un algorithme de classification par apprentissage supervisé nommé Support Vector Machine (SVM) est utilisé pour associer les tailles de paquets à un type d'application. Pour la réalisation, nous utilisons des cartes de gestion de trafic intégrant des FPGAs comme la NetFPGA 10G.

Nous avons d'abord construit un algorithme simple et efficace de stockage de flots inspiré de l'algorithme CMS. Il supporte avec quasiment aucune perte le stockage d'un million de flots simultanés, et garantit des temps de recherche et d'insertion constants. Il utilise aussi assez peu de ressources sur le FPGA.

Le processus de classification de flot basé sur SVM est réalisé de manière complètement parallèle grâce à un pipeline capable de commencer le calcul sur un nouveau vecteur du modèle SVM à chaque coup d'horloge. Pour accélérer ce processus, plusieurs unités de calcul peuvent fonctionner en parallèle, divisant le temps requis pour classifier un flot.

Le noyau est une fonction importante de l'algorithme SVM. Nous en avons testé deux. Le classique RBF, et une version plus adaptée à une plateforme matérielle que nous avons nommée CORDIC. Les deux noyaux offrent la même précision, mais le CORDIC supporte des fréquences de fonctionnement plus élevées et utilise moins de place sur le FPGA, ce qui permet de mettre plus d'unités de calcul en parallèle. Le modèle SVM obtenu avec le noyau CORDIC est plus simple et plus rapide à gérer que le modèle obtenu avec RBF, ce qui apporte une amélioration inattendue du temps de traitement.

Grâce à ces optimisations, la classification de flots peut être faite à 320 075 flots par seconde pour un modèle SVM avec 8 007 vecteurs, ce qui permettrait une classification de la trace la plus réaliste que nous avons testée, Brescia, à 473 Gb/s. Pour augmenter la vitesse supportée par le classificateur, différents paramètres devraient être changés :

- utiliser des interfaces réseau à plus haut débit ;
- utiliser un meilleur FPGA pour permettre de paralléliser SVM encore plus ;
- utiliser une plus grosse mémoire externe pour supporter plus de flots simultanés sans augmenter le risque de perte.

Une autre amélioration possible de la version actuelle serait de la rendre plus

flexible en stockant le modèle SVM dans des RAMs plutôt que des ROMs, pour éviter d'avoir à refaire une synthèse pour changer le modèle.

Le mécanisme de stockage de flots peut être réutilisé pour toute application réseau basée sur les flots. Le code de SVM sur FPGA est générique et peut être utilisé pour d'autres problèmes de classification.

Cette réalisation sur FPGA d'un algorithme de surveillance de trafic rend les avantages et inconvénients de cette plateforme visibles :

- le développement est long et complexe. Des problèmes comme la quantification des variables doivent être pris en compte. D'autres problèmes triviaux en logiciel, comme le stockage des flots, doivent être gérés manuellement à cause des ressources limitées.
- La flexibilité n'est pas automatique. Par exemple de nouveaux développements seraient nécessaires pour changer dynamiquement le modèle SVM.
- L'accélération est très conséquente, surtout pour des algorithmes comme SVM qui peuvent être massivement parallèles.
- Il est facile de garantir un traitement en temps réel car chaque délai est connu et aucune autre tâche ne peut interférer avec le traitement en cours.

B.5 Plateforme de test avec accélération matérielle

Nous avons vu deux approches de surveillance du trafic. Une purement logicielle et une purement matérielle. Dans chaque cas, nous avons développé des prototypes fonctionnant à 10 Gb/s. Pour les tester, nous avons eu besoin d'un générateur de trafic. Nous avons utilisé un générateur commercial, mais nous souhaitons maintenant créer notre propre générateur, plus flexible et meilleur marché. Nous voulons un générateur qui supporte au moins 10 Gb/s sans aucun paquet perdu, même avec le trafic le plus dur à gérer. C'est pourquoi nous le développons sur FPGA. Mais nous voulons aussi une grande flexibilité, d'où l'utilisation d'une approche hybride logicielle et matérielle.

Le générateur de trafic est open-source. Il utilise le FPGA d'une carte Combo pour saturer deux interfaces à 10 Gb/s facilement, même avec les plus petits paquets permis par le protocole Ethernet. L'architecture modulaire du générateur lui permet d'être flexible à tous les niveaux.

La façon la plus simple de personnaliser le trafic généré est d'utiliser l'interface graphique de configuration. Quelques clics suffisent pour spécifier le trafic sous la forme de flux de paquets qui partagent certaines caractéristiques. Chaque flux peut atteindre 10 Gb/s. L'utilisation de plusieurs flux concurrents permet de générer un trafic diversifié, ou d'envoyer du trafic sur plusieurs interfaces.

Contrairement aux générateurs de trafic commerciaux, si l'interface graphique n'offre pas les bonnes options pour générer le trafic voulu, l'utilisateur peut développer ses propres fonctions. Le générateur est fait pour simplifier l'ajout de nouveaux modules appelés modificateurs. Le développement d'un modificateur

nécessite de connaitre le VHDL et un peu de Python, mais cela reste aussi simple que possible. Tous les modificateurs existants sont documentés et peuvent être utilisés comme exemples.

Bien que le générateur fonctionne pour le moment avec la carte Combo d'Invea-Tech, cette carte est très similaire au NetFPGA 10G, bien connu des chercheurs. La plateforme NetCOPE utilisée par le générateur est compatible avec le NetFPGA 10G, donc porter le générateur vers NetFPGA devrait être assez simple. Nous souhaitons le faire dés que possible.

Actuellement, le générateur de trafic est fait pour générer du trafic à haut débit pour des tests de charge. Il n'est pas adapté pour générer du trafic réaliste. Mais cela pourrait changer simplement en développant de nouveaux modificateurs contrôlant les tailles et délais inter-paquets selon l'un des nombreux modèles de trafic disponibles dans la littérature.

Comme le générateur de trafic est un projet open-source, son code est disponible en ligne [Gro13]. Si certains lecteurs sont intéressés par le développement de nouveaux modificateurs ou veulent aider à porter le générateur vers NetFPGA, ou s'ils ont des idées pour améliorer le projet, qu'ils n'hésitent pas à s'impliquer.

L'architecture du générateur de trafic est un bon exemple de l'utilisation des avantages des FPGAs sans perdre la flexibilité pour l'utilisateur. Les FPGAs apportent des avantages clairs pour le contrôle du temps réel. Le développement bas niveau permet de contrôler les temps inter-paquets bien plus facilement qu'en logiciel. Le support du débit maximum se fait naturellement grâce au parallélisme du FPGA. Une fois que l'architecture est conçue correctement, le support du débit maximum est garanti.

Mais le développement sur FPGA est long et complexe, il devrait donc être évité autant que possible. Faire communiquer le FPGA avec l'ordinateur pour recevoir les données de configuration est une bonne manière de garder une partie du traitement en logiciel. Du point de vue de l'utilisateur, bien que le FPGA soit un outil bas niveau, il peut être caché derrière une interface graphique simple et utilisable par tous. Quand le développement sur FPGA est nécessaire malgré tout, il peut être simplifié en définissant clairement l'architecture dans laquelle les nouveaux blocs doivent être intégrés, et en fournissant des exemples.

B.6 Conclusion

Cette thèse propose différentes méthodes pour accélérer la surveillance de trafic pour supporter de très hauts débits. La meilleure solution dépend de l'application et de ses besoins les plus importants. L'accélération doit être prise en compte à la fois au niveau algorithmique et au niveau du développement pour obtenir les meilleurs résultats. Nous avons vu qu'une bonne manière d'obtenir à la fois les avantages du matériel (haut niveau de parallélisme, contrôle du temps réel) et du logiciel (facilité de développement, flexibilité) est de concevoir des architectures mixtes. La partie traitant directement les paquets est gérée par le matériel, et la partie qui ne traite que des données accumulées est gérée par le logiciel.

De nombreux outils haute performance développés pour nos prototypes peuvent être réutilisés dans d'autres applications : le comptage de paquets utilisant l'al-

gorithme CMS (logiciel), la détection de changements dans des séries temporelles utilisant l'algorithme CUSUM (logiciel), la classification de données utilisant l'algorithme d'apprentissage SVM (matériel), le groupement de paquets en flots utilisant un algorithme inspiré de CMS (matériel), la génération de trafic de manière flexible et efficace (matériel).

Ces développements permettent de déduire quelques règles utiles pour choisir une plateforme de développement pour un application de traitement de trafic :

- Le débit maximal supporté est le point faible des implémentations logicielles. Un débit de 10 Gb/s peut être un problème dans le pire cas, même pour une application simple.
- Le développement sur FPGA est un processus long et complexe, ce qui rend le développement logiciel plus attrayant.
- Cependant supporter des débits élevés en logiciel nécessite un développement précis et de très nombreux réglages de la plateforme, qui peuvent ralentir fortement le développement logiciel.
- Les architectures hybrides logicielles et matérielles peuvent être très efficaces, à condition de faire attention au débit de communication avec le processeur.

La possibilité d'intégrer un FPGA et un processeur sur la même puce, ou au moins sur la même carte, permettrait une communication plus facile entre les deux et des possibilités d'interactions plus grandes.

Les besoins en surveillance de trafic semblent se multiplier actuellement car les gouvernements souhaitent de plus en plus contrôler ce que les gens font sur Internet. Un Internet sans contrôle ne pourrait pas fonctionner, mais la surveillance doit prendre en compte le respect de la vie privée des utilisateurs. En effet, malgré l'utilisation de systèmes de chiffrement, il est techniquement possible d'apprendre énormément de choses sur les personnes utilisant le réseau grâce à des systèmes de surveillance en temps réel tels que ceux présentés ici.

Chapter 1

Introduction

1.1 Context

Traffic monitoring is a mandatory task for network managers, be it small company networks or national Internet providers' networks. It is the only way to know what is happening on a network. It can be very basic and consist in simply measuring the data rate of the traffic on a link, or it can be more sophisticated, like an application to detect attacks on protected servers and raise alarms.

Use cases for traffic monitoring are diverse and can be separated into different categories. An important category is security. Networks can be used for all kinds of malicious behaviours: for example propagating viruses through malicious websites or emails, sending unsolicited emails, scanning the network to find vulnerable machines, taking remotely control of machines without authorization, sending a large amount of traffic to a machine to overload it, or intercepting traffic to access to sensitive data. Most of these behaviours produce specific traffic that may be identified using traffic monitoring tools. The challenge is that legitimate traffic can take many forms, making it difficult to differentiate the malicious traffic. And attackers do all they can to make the malicious traffic look legitimate, so attacks and monitoring tools have to become more and more elaborate.

Many attacks have made the news lately, like an attempt to take the NASDAQ infrastructure down [Rya13]. These attacks are often successful and outline the need to have monitoring tools that can detect attacks early and make it possible to mitigate them efficiently. Attackers use multiple techniques to create successful attacks. First they hide their identity. To do that, they never conduct attacks directly. They take control of machines that are not protected properly, and use them to conduct the attacks. They also often hide the origin of the traffic by indicating fake source addresses. Techniques used to take control of machines usually take advantage of flaws in legitimate applications and protocols. So security has to be applied at multiple levels. The first thing to do is to fix flaws as soon as they are discovered. Antivirus software can also be used to protect individual machines by detecting applications with unusual behaviours. Firewalls can restrict incoming and outgoing traffic to try to prevent attackers from accessing to sensitive machines. At the network level, providers can stop traffic with fake source addresses only if it comes directly from their own customers. As

many providers do not apply such a strategy, fake source addresses cannot be fully avoided. But network providers can also apply more advanced algorithms to monitor the traffic and detect specific traffic if it behaves like an attack. They can then stop the traffic early, making the attack fail. This kind of protection is not really offered by providers yet, but solutions exist in the literature.

Another category of traffic monitoring applications is used for traffic engineering. It mostly consists in getting information about the traffic transiting in the network or the state of the network. An important application is traffic classification, which is able to associate traffic to the application that generated it. On the Internet, traffic is made of an aggregation of packets, each packet represents a piece of data. Each packet has a header giving some information like an address for the sender and the receiver of the packet. The address of the receiver is used to route the packet in the network. But there is no standard way for a packet to indicate information like the application that generated it, or the kind of data it contains. The role of traffic classification information is to use the packet data and header to guess the application or the category of applications that generated the packet. For example a packet has been generated by Skype and contains voice data. Another packet has been generated by Firefox and contains web data. A packet could also be generated by Firefox but contain video streaming data, when browsing YouTube for instance.

Traffic classification can be useful to network managers in different ways. It can be used to prioritize some traffic. For example packets that contain videoconference data must be delivered fast, so as to make the conversation fluid. Packets that contain web data can be delayed if necessary, because it will only delay the display of the page of some milliseconds. Another use of traffic classification is lawful interception: governments are more and more interested in forcing network operators to log some specific traffic, like Skype calls for example. Some use cases make traffic classification more challenging than others. For example, if it is used to prioritize traffic, some users may try to make all their traffic resemble important traffic, so malicious behaviour should be taken into account.

Be it for security or for traffic engineering, the need for automatic accurate monitoring tools is rising. This is partly due to a current trend in network management called software-defined networking. It consists in decoupling the control and data planes of a network. Traffic is managed by manipulating abstractions that are not directly related to the physical network. The goal is to make networks more flexible: creating new services or adapting to new constraints becomes just a matter of software configuration. To work, this approach requires flexible and accurate network monitoring tools to be able to know what is happening on the network and to check the impact of global configuration changes.

Although traffic monitoring applications can be diverse, they face common challenges. The most important is due to the evolution of the Internet. CISCO forecasts [Cis13] that between 2012 and 2017, the global Internet traffic will increase threefold. It will reach one zettabytes per year in 2015. This means that network operators have to cope with always higher data rates. To do that, they deploy links supporting higher data rates. Traffic monitoring applications are not deployed directly on very high-speed links like transatlantic cables that can have

a speed of multiple terabits per second. But aggregation links of 10 Gb/s or more are common, as operators now easily offer 1 Gb/s to their customers with optical fiber.

Supporting links of 10 Gb/s or more is difficult because it forces to process millions of packets per second. Current commercial computers are not offered with 10 Gb/s Network Interface Cards (NICs), but even if they were, they would not be able to process that many packets. This is why high performance developments are necessary. Different platforms are available for high-speed traffic processing. The most common is commodity hardware: powerful commercial computers can be used, fine-tuning software and using specialized NICs to make them support higher data rates. Graphics Processing Units (GPUs), which are originally designed for gaming, can also be added to perform highly parallel computations. But other platforms exist that provide more specialized hardware acceleration. Some special processors, called Network Processing Units (NPUs), have an architecture designed to handle high-speed traffic. At a lower level, Field-Programmable Gate Arrays (FPGAs) are chips that can be configured to behave like any digital electronic circuit. They provide massive parallelism and a very low-level access to network interfaces. Each platform has its own advantages and drawbacks in terms of performance, development time, cost and flexibility.

1.2 Objectives

The subject of this thesis is to evaluate different ways to accelerate traffic monitoring applications. A large body of literature exists in the domains of security and traffic engineering to tackle classical problems like anomaly detection or traffic classification. The accuracy of the proposed algorithms is usually well studied, but less work exists about the way an algorithm can be implemented to support high data rates.

To accelerate traffic monitoring applications, we will test different development platforms that provide software or hardware acceleration and see how they are adapted to traffic monitoring. We will implement applications for security and traffic engineering on different platforms, trying to take advantage of all features of the chosen platform, and to avoid the drawbacks specific to the platform.

The first application in the domain of security will use pure software, leveraging the flexibility inherent to software development, so that lightweight security algorithms can be used in many different contexts. It will allow to check the performance computers can provide using optimized software. The second application in the domain of traffic engineering will use an FPGA to accelerate an algorithm making heavy computations, so as to meet strong real-time requirements. It will show that hardware acceleration can bring huge speed improvements to traffic monitoring, at the cost of some other criteria like development time and flexibility. The last application, a traffic generator, will be a trade-off between flexibility and performance, using an FPGA to meet real-time requirements, as well as software to provide a simple configuration tool.

These different use cases will showcase all the aspects that should be taken into account when building a high-speed network processing application, like the

performance, the development time, the cost or the flexibility. We will not test NPUs or GPUs because they require some very specific developments, and the advantages they bring (parallelism, accelerated functions, specialized architectures) can also been obtained on an FPGA. So FPGAs are a good choice to explore all possible acceleration techniques.

But to accelerate an application, it is not enough to work on an efficient implementation of a pre-defined algorithm. We will see that some algorithms are more adapted than others to support high data rates, because they are easier to parallelize or because they use less memory than others. We will also see that tweaking some algorithms can widely improve their scalability without necessarily hindering their accuracy.

Finally, we will be able to give advices on the development platform that should be chosen depending on the application, and we will provide some efficient implementations of useful traffic monitoring applications. Many developments will be reusable for other applications too, either as libraries for specific tasks, or as fully functional products.

1.3 Traffic monitoring

Before starting to implement traffic monitoring applications, it is important to choose the right algorithm. To do this, we will list some important features of these algorithms, focusing on the ones that impact strongly the implementation. Similar algorithms will result in the same implementation challenges.

We will discuss the **topology** of the application, that is to say the place in the network where the probe or probes are located. The **time constraints** on the algorithm are important too: some processing has to be done fast, while other things can be delayed. The **traffic features** used define the way received packets will be treated. The **detection technique** then defines how gathered data will be processed. Some applications also require a **calibration**, which is usually a separate implementation from the core of the application.

1.3.1 Topology

The first thing to differentiate traffic monitoring applications is the way they are located on the network. The simplest alternative is to have only one standalone probe on a link. The advantage is that it is easy to setup. It requires only one machine and it is easy to install for a network operator.

But many traffic monitoring applications are distributed. They are made of different probes on different links on the network. This way, the application has a more global view of the activity on the network. Often collectors are added to the probes. They are machines that do not monitor the traffic directly but centralize reports from the probes. They can also analyze and store data. Distributed applications are more costly to setup. and the communication protocol must be well designed to avoid exchanging too much control data when monitoring traffic at high data rates.

Distributed applications are usually deployed on one network, with all probes belonging to the same operator. But it is also possible to create inter-domain traffic monitoring applications, with probes on networks from different operators. This is interesting because the application has an even more global view of the network. The main difficulty is that operators are usually reluctant to share data with potential competitors. But it is possible to let operators share some aggregated data with operators if they are part of the same coalition.

The chosen topology fully changes the contraints on the implementation. Standalone applications may have to cope with very high data rates, and they have to take care of the whole processing. So they need a lot of processing power. Distributed applications can use multiple machines for the processing. Each probe may receive less traffic as it is distributed on multiple probes. But communication delays between the machines can be long, and the amount of data sent should be limited to avoid overloading the network. Inter-domain applications often have stronger security requirements as data is transmitted through less trusted networks.

1.3.2 Time constraints

Time constraints are the requirements for the application to finish a certain processing within a guaranteed delay. Depending on the application, time constraints can be very different. For example an application monitoring the data rate on a link may log it only every minute. The delay is not a problem. But an application that detects attacks must raise an alert fast so that the network administrator acts as fast as possible. And it is even worse for an application, which directly acts on the transiting traffic. For instance, a traffic classification application can tag packets on a link with a label depending on the kind of data transported. To do that, the application has to process packets very fast. If it is slow, it will delay all packets on the link, degrading the quality of the communication.

Time constraints must be taken into account when implementing an algorithm. Some development platforms are more adapted than others to real-time applications. Most applications are made of different parts that must meet different time constraints. For example, the part directly receiving packets must work in real-time, but another part gathering data and storing it periodically in a database can be slow.

1.3.3 Traffic features

Traffic monitoring applications can be divided into two categories: active measurement applications and passive measurement applications. For active measurements, applications send some traffic and then observe how it is managed by the network. This synthetic traffic can be observed by a different probe somewhere else on the network, or an answer from another machine can be observed directly by the same probe. Examples of passive measurements can be a delay for packets to be sent from one probe to another, or a delay for a machine to answer to a request from the probe. The absence of answer can also be informative.

For passive measurements, applications extract data from different features of the traffic. Some simply use data in packet headers, like the size of packets. Others use the whole data of the packet. Many applications base their analysis on flows. The structure of packets is specified by encapsulated protocols. The lowest level protocol is usually Ethernet, it allows machines to communicate when they are directly connected. Then the usual protocol is Internet Protocol (IP), it allows machines to communicate, even if they are not directly connected. The IP header contains the source and destination IP addresses, unique identifiers of the sender and the target. Then two transport protocols are very common: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). They both are made to help applications communicate with each other. To differentiate applications on the same machine, they define source and destination ports. An unidirectional flow is a set of packets with the same source and destination IP address, the same transport protocol (UDP or TCP) and the same source and destination port. Some applications are based on bidirectional flows, that is to say that packets with inverted sources and destinations belong to the same flow. But most traffic monitoring applications are based on unidirectional flows because there is no guarantee on a network link to see the two directions of a flow: packets can be sent using one path on the network, and come back using another path.

Some applications are based on other features. For example they can analyze only Domain Name Service (DNS) requests, that is to say packets sent by users to ask for the IP address corresponding to a domain name. It is an easy way to list the servers to which each user connects. It is also possible to analyze only Border Gateway Protocol (BGP) packets. They are used by routers to communicate data about the IP addresses they know how to reach. Analyzing these packets can help have a clear view of the current network routing.

Higher-level approaches exist, like behavioural analysis: communications on the network are studied to determine who communicates with whom and how. A graph is then built and the communication pattern of each machine can help identify the role of the machine.

The traffic features used impact the implementation because some are easier to extract than others. A passive probe extracting only some headers from packets is easy to implement. Active probes measuring delays need an accurate way to measure time, which is not available on all platforms. Flow-level features force the probe to group packets of the same flow, storing data about each flow in a memory. Constraints on the memory are strong: it has to be big enough to store all flows, and the read and write speeds have to be fast enough to process received packets in real time.

1.3.4 Detection technique

All traffic monitoring applications use an algorithm to translate traffic features into information. This is what we call the detection technique, although the word detection may not be adapted to all applications.

Many applications use signatures: signatures of the applications for traffic classification, or signature of an attack for security. A signature is a set of features.

For example it can be a sequence of bits that has to be found at a certain location in a packet. If some traffic corresponds to this set of features, it corresponds to the signature. This technique is widely used for traffic classification because applications tend to generate always the same kind of traffic.

Learning techniques are also often used for classification. They consist in building a model for each type of traffic to detect. The model is built by observing traffic for which the category is known. When new traffic arrives, computations are made to check to which model it belongs the most.

Anomaly detection is another technique, widely used for security. It requires a knowledge of the normal traffic. When some abnormal features are detected, an alarm is raised because it is a potential attack. The advantage is that attacks that never happened before can be detected this way.

Some detection techniques can be very simple to implement: tree-based classification is usually implemented by a simple succession of conditions. Other techniques can require complex computations from the traffic features, making the implementation more complicated, and the resulting application slower. Some techniques are also more adapted to take advantage of hardware acceleration than others, so the choice is crucial.

1.3.5 Calibration

Most traffic monitoring applications require a calibration to be functional. The calibration process depends on the detection technique. It is how signatures or models are created. For learning techniques to build their models, three methods exist:

- Supervised algorithms need a "ground truth", an example of traffic with data already attached. For example in the security domain, attacks should be identified in the trace. For traffic classification, the generating application should be identified.

- Semi-supervised algorithms need a ground truth too, but it can contain unknown traffic. The algorithm will guess to which category the unknown traffic should belong, and include it in its learning.

- Unsupervised algorithms require no ground truth, which makes them very easy to deploy. Often their output is less detailed than supervised algorithms.

For techniques based on signatures or anomalies, a ground truth is often necessary too. The calibration is often more manual than for learning algorithms. An expert can extract signatures from a ground truth, using its own knowledge of the traffic. An expert is also often required to define rules specifying the behaviour of normal traffic, so as to be able to detect anomalies.

The calibration does not directly impact the implementation because it is usually done offline with no timing constraints. But it deeply impacts the ease of deployment of the solution. An algorithm which requires no ground truth is simple to deploy and will probably work the same way on any network, while

an adapted ground truth will have to be found for each deployment for other algorithms. But using a ground truth is a way to extract more accurate data about the traffic.

1.4 Acceleration challenges

We have seen that traffic monitoring applications have common features. This is why one is often faced with the same challenges when accelerating a traffic monitoring application to support high data rates. We already talked about high-level challenges like the need to receive data at high speed, the need to make fast computations and the need for flexible applications. But some less obvious challenges are limiting factors found in most high-speed monitoring implementations. We will now present two of these challenges, which are not obvious but will be found multiple times in next chapters. The solutions provided in next chapters can often be reused for other traffic monitoring applications.

1.4.1 Large data storage

Traffic monitoring implies handling huge amounts of data. For example, dumping all traffic on a 10 Gb/s link during one minute requires 75 GB of disk space. The challenge becomes even bigger if data has to be stored in a way that is simple to reuse afterwards. A simple task like counting the number of packets sent to each IP address seen by a network probe requires to maintain one counter (an integer of 32 bits for example) for each destination IP address. If Internet Protocol version 4 (IPv4) is used, $2^{32} = 4.3 \times 10^9$ addresses exist. So using a simple memory with the IP address used as memory address would require $2^{32} \times 32 = 1.4 \times 10^{11}$ bits, that is to say 17 GB of memory. This is even worse if we want to count packets with the same source and destination IP addresses, we would need 7.3×10^7 TB of memory.

The realistic and classical solution in computer science is to use hash tables [SDTL05]. These storage structures are able to store elements in memory indexed by a key (the IP address in our example). The space they require in memory is the size of stored data plus the size of the key for each element. The key is stored alongside the pieces of data it indexes for future references. So for example if one million IP addresses are seen, a bit more than $1\,000\,000 \times (32+32)$ bits will be required to store the counters. The first 32 bits are for the IP address and the others are for the counter itself. Hash tables provide very low mean delays for accessing or updating data from the key. The only problem is that to make sure that a new element is always stored somewhere, access and update delays can sometimes become very long, as long as the time needed to browse the whole list in the worst case. This is a blocking problem for real-time monitoring applications that must guarantee that they are able to support a certain data rate at all times. Another drawback is that the memory used can expand over time to make space for new data to store, when new IP addresses are discovered. Depending on the application, having no guaranteed maximum memory requirements can be a problem.

This is why data storage often is a challenge for network monitoring applications. This is especially true when available memory is constrained, as we will see it is the case on some hardware-accelerated network monitoring platforms. This challenge will arise when trying to count packets by IP address for an attack detector in Section 3.2.2. It will also arise when trying to keep data about flows for a traffic classifier in Section 4.4.2.

1.4.2 Test conditions

Once implemented, traffic monitoring applications have to be tested to validate their performance in terms of accuracy and in terms of supported data rate.

To validate the accuracy of an algorithm, realistic traffic should be used. The best way to do that is to use a trace of actual traffic. The problem is that only network providers have this kind of traces, and they do not particularly like sharing them. Some traces are freely available though. But to be interesting, the trace should be accompanied by a ground truth, just like the traces used for calibration. Otherwise there is no way to tell if the traffic monitoring application works properly or not. So whatever the application, test traces are always very challenging to find.

To validate the supported data rate of an application, the best way is to use synthetic traffic that is as hard to handle for the application as possible. To do that, a configurable traffic generator is necessary. A simple computer cannot be used because the generator must be able to reach data rates at least as high as the application under test. Commercial traffic generators exist but they are expensive.

This challenge will arise in chapters 3 and 4 to test the algorithms we propose. Chapter 5 will bring a partial solution by describing the implementation of an affordable high-speed traffic classifier.

1.5 Thesis structure

Chapter 2 is about the different development platforms available to accelerate traffic monitoring applications. It lists different criteria that should be taken into account, and uses examples from the literature to analyze the advantages and drawbacks of each development platform. It is the chapter to read before choosing a development platform for a new traffic monitoring application. Following chapters base their platform choices on this one, and they allow to experiment to quantify the limits of each platform.

Chapter 3 is a first traffic monitoring application implementation on the most widespread development platform: commodity hardware, that is to say normal computers. It is realized in the framework of a European FP7 project called DEcentralized, cooperative, and privacy-preserving MONitoring for trustworthinesS (DEMONS). We participated to the development of one of the important project outcomes: a flexible high-performance software platform for traffic monitoring called BlockMon. We developed an application for attack detection to test the

flexibility and the supported data rate of BlockMon. The result is a very flexible platform that supports up to 10 Gb/s, which is good for a pure software application, but the application reaches the limits of commodity hardware.

This is why Chapter 4 is focused on the use of hardware acceleration on FPGA. The implemented application is not in the security field, but in the traffic engineering field. It is a traffic classifier. This application has been chosen because it uses a learning algorithm that is both challenging and interesting to implement using hardware acceleration. The algorithm and the implementation are designed together to get the best performance using the possibilities of FPGAs as much as possible. The results show the huge increase in performance due to hardware acceleration. The application also outlines a drawback of hardware development: it is more difficult to provide flexibility.

Chapter 5 describes the implementation of a traffic generator using hardware acceleration on FPGA. The need to develop a high-speed, accurate and flexible generator arose when we tested our monitoring applications. As we think this generator can be useful to the Research community to test all kinds of traffic processing applications at high speed, it is fully open-source. It provides a way to generate accurate and diverse high speed traffic without using extremely expensive commercial traffic generators. The generator is implemented on the same platform as the traffic classifier. It is able to send traffic up to 20 Gb/s, and we would like to port it to a different platform to generate traffic up to 40 Gb/s. During development, we decided to address a drawback of hardware accelerators, outlined by our traffic classifier implementation: the flexibility. The generator is configurable using a simple Graphical User Interface (GUI). It is also easily extensible to add new features.

Finally, Chapter 6 summarizes our contributions. Based on the different implementations we have made on different platforms, it draws conclusions about the way each platform can be used depending on the complexity of the algorithms as well as the most important requirements on the application. It gives a short guide on how to make design choices for a traffic monitoring application at the algorithmic and implementation level, and presents some perspectives.

Chapter 2

Choosing a development platform

Although traffic monitoring applications can be very diverse, one is always faced with similar challenges when implementing them. Different development platforms are capable of handling traffic monitoring: cheap platforms made from commodity hardware, specialized platforms using network processors, or hardware-oriented platforms using FPGAs. Each option has different advantages and drawbacks. The intent of this chapter is to help make the good choice depending on the application specifications. It will first list the criteria to base a choice on, then describe each platform with its strengths and weaknesses.

2.1 Criteria

All platforms have in common a network interface able to receive and send traffic simultaneously, typically on one or more Ethernet links. They also have one or more processing units, which can receive data from the interface, make various computations and send data to the interface. The processing units are programmed or configured by the developer.

The differences are due to the kind of processing units, the way they communicate between each other, and the way they communicate with the network interface. Different processing units provide different levels of parallelism and communicate more or less easily with the network interface. The development complexity is also different. This makes their strengths vary for the criteria listed below, which can allow to choose one platform instead of an other.

2.1.1 Supported data rate

This is the most obvious feature common to all traffic monitoring applications. Independently of the application, the development platform has a maximum supported data rate. On the receiving side, it is the speed at which it can receive data and make it ready for processing, without any actual processing happening. On the sending side, it is the speed at which in can send data that is already ready to send. The most important is the sustained rate, that is to say a rate that can be supported indefinitely.

There is no guarantee that an application developed on a given platform will support the maximum rate, because the processing part can be the bottleneck. But no application can support more than the maximum rate of the platform.

This data rate is not always the speed supported by the network interfaces, for example 10 Gb/s for 10 gigabit Ethernet. It is not enough to receive data on the interface: it must be made available to the processing units. The maximum data rate should be supported for any kind of packet. On Ethernet, this is usually more challenging for the smallest packets (46 bytes of payload) because a processing overhead is associated to each packet.

2.1.2 Computation power

Once the packets are received and sent fast enough, they must be processed fast enough too. This is the role of the processing units. There can be many of them of different kinds: a Central Processing Unit (CPU), a GPU, a NPU or an FPGA. Each kind of processing unit provides processing power with important differences in:

- the frequency: all processing units work using a discretized time, the frequency is the number of clock cycles that can be run during one second;
- the parallelism: the number of operations that can be run in parallel during one clock cycle;
- the variety and complexity of each operation.

Depending on the application, the need in computation power can vary widely. If the goal is only to count the number of received packets and bytes, to estimate the quantity of data received, there is almost no computation needed. If the goal is to classify traffic using Deep Packet Inspection (DPI), an important number of regular expressions have to be checked against each packet [AFK+12], which requires a lot of computing power.

2.1.3 Flexibility

The computation power itself is not enough to quantify the acceleration factor a platform will bring to a specific application. Depending on the task, the development platform that will bring the most acceleration is not always the same.

For example, one platform may have very fast memory accesses, which will make it very good at handling large lookup tables, while another might have very fast floating-point computation units, which will make it adapted to complex mathematic operations.

This is particularly true for platforms which are specialized for certain tasks, and have highly optimized units dedicated to these tasks. An application relying heavily on these tasks will be a perfect fit for this platform, but some applications may not use these tasks at all.

So it is important to identify which tasks might become bottlenecks throttling the maximum supported data rate of an application. Development platforms

should be especially evaluated on these tasks. If the application is not very defined yet, the best choice will be a less specialized platform.

2.1.4 Reliability

To run on a real network, a traffic monitoring application has to be reliable. This can have different aspects:

- The supported data rate of the application must be reliable: for example, if the computation speed depends on external factors (other applications on the same machine), this can slow down the packet processing and lower the supported data rate during certain periods of time.

- The measurements made on the received traffic must be reliable. This is particularly a problem for the timestamp that indicates when a packet was received. Some applications rely on this delay, like applications that measure the Quality of Service (QoS) on a network [OGI+12], or some traffic classification applications [DdDPSR08]. If the time to process a packet is variable and unknown, the timestamp will be impossible to obtain accurately.

The reliability is mainly linked with the ability for the processing units to support real-time applications, that is to say applications that must perform some computations in a determined delay.

2.1.5 Security

No development platform can guarantee the security of any application. But it is easier to develop secure applications on some platforms. A perfectly secure application would work reliably even if attackers try to prevent it, and would not expose data that should remain private.

For some applications security may be crucial. A firewall is for example an obvious target to attack a network. For other applications, like passive probes that only collect statistics about the traffic, security is not the most important feature.

2.1.6 Platform openness

This factor is often overlooked, but the openness of a platform has an important impact on the development process. Different parameters can make a platform more or less open:

- The availability of open-source projects using the platform. Studying a well-documented open-source project is a very easy way to learn how to develop for a specific platform. The presence of an online support community is a big help too [SH13].

- The possibility to study and modify the code of the development framework. If this code is open-source and well documented, developers will be able to understand faster how to develop new applications.

- The possibility to reuse the code developed for this platform on other similar platforms. It may be important to not be locked with specific hardware or a specific vendor, so as to be able to change for a new cheaper of more powerful option. This depends mainly on the language and framework used. Some frameworks are focused only on working on a specific platform, while others are focused on inter-operability. This difference can be seen for example in frameworks for development of scientific applications on GPUs: CUDA works only on Nvidia GPUs, while OpenCL is made to work on all GPUs and on other kinds of processing units like FPGAs [DWL+12].

2.1.7 Development time

The development time is especially important for research applications: testing different algorithms will require to implement them all on the same platform. For a production application, the algorithm is chosen in advance. If the risk of changing requirements is not too big, a platform with a long development time may be chosen if it brings better performance.

A development process is divided in three phases that can be repeated: design, development and test. The development and test times strongly depend on the platform. For example, the availability of high-level functions adapted to the application will speed-up the development considerably. It can also speed up the tests because the provided functions are already tested. The time to test is also strongly affected by the efficiency of the debugging tools.

2.1.8 Update simplicity

In production, most applications will need to be configured or updated. This process may be more or less complicated depending on the development platform:

- modifying the application and getting a new deployable version can take a long time if the test process or a compilation process is long,

- a physical handling of the machine may be required, making an online update impossible,

- the unavailability time of the application during the update can vary. Depending on the real-time needs of the application, this point may become crucial.

2.1.9 Future scalability

The future scalability depends on the amount of work that would be required to adapt an existing application to support higher data rates. Most platforms evolve, and more powerful hardware becomes available. The exploitation of these

improvements requires changing the application. The complexity of this process depends on the considered application. But the design, the code modularity and the abstractions that certain platforms enforce help make these changes easier.

For example, to exploit higher parallelism, most platforms will allow to run more concurrent threads of the same code simply by changing a parameter and compiling the application again. Some might even decide of the best level of parallelism at run time.

2.1.10 Hardware cost

The hardware cost is a very visible criterium. This is the first encountered cost when starting with a new development platform. Specialized hardware will cost more than commodity hardware, but depending on the scale of the deployment, different costs must be considered:

- If the application will be deployed only in few places on a large network, the hardware cost will remain low compared to the development cost, even choosing expensive options.
- If the application will be deployed in thousands of places, the hardware cost becomes important. Choosing the most affordable option will probably lower the total cost.
- If the application will be deployed in hundreds of thousands of places, mass production becomes interesting. In this case, a solution with a high development cost but a small unitary hardware cost is the best choice.

If the goal is only to develop a prototype, a flexible development platform is needed, even if it is expensive. But it is important to make the transition to cheaper hardware for mass production easy.

2.2 Commodity hardware

When developing a traffic monitoring application, the most obvious idea is to use commodity hardware that is often already available. NICs are now publicly available that are able to handle up to 40 Gb/s of traffic [Int13]. And if the CPU is not powerful enough to run the algorithm, a GPU may be used. This section will describe the principles and challenges of using commodity hardware, and will evaluate this development platform using criteria from Section 2.1.

2.2.1 Handling traffic

To receive and send Ethernet packets using a computer, a NIC is used. For example, Intel has a NIC with four interfaces at 10 Gb/s [Int13], at a **hardware cost** of about $1 100. To allow the host CPU to handle multiple received packets in parallel, this NIC has a feature called Receive Side Scaling (RSS). This technology divides the received packets into multiple queues, which are sent concurrently to

the host CPU. This way, CPUs with multiple cores can handle multiple packets in parallel. The rule to decide which packet is sent to which queue is configured by the developer. It is based on the value of a hash computed on some configured fields of the packet headers.

But connecting a modern NIC like this to any computer is not enough to support 40 Gb/s incoming and outgoing traffic in an application. Usually Linux is used for network monitoring applications development, but the Linux kernel, which handles the communication with the NIC, was not designed to support 40 Gb/s interfaces.

The current Linux kernel already takes advantage of some improvements that help supporting higher data rates:

- Multiple hardware queues (RSS) support has been added. This means that if the NIC supports the RSS technology, packets can be received concurrently on multiple cores.
- New Application Programming Interface (NAPI) has been introduced. This is a new way for the Linux kernel to handle received packets. It is based on two main principles [SOK01]:

  Interrupt mitigation is a way to avoid receiving one interruption per packet, which would overload the CPU. When the threads managing the packets are busy, no more interruptions are sent.

  Packet throttling is simply a lower-level dropping of the packets which cannot be received when the CPU is overloaded. This means that less work is needed to handle these lost packets.

But this is not enough to support 40 Gb/s on commodity hardware. Different papers present factors that limit the supported data rate in the design of the Linux kernel [BDPGP12b, RDC12, LZB11]:

Inefficient memory allocations. For each received packet, some memory is allocated by the driver to store a packet descriptor. It is then de-allocated when the packet is sent back to memory. This process is slow, and takes a constant time for each packet. This means that it is especially a problem with small packets, reducing the maximum data rate.

Communication between the kernel and applications. For security reasons, the memory used by the kernel is different from the memory used by applications. This means that data from the received packets must be copied twice in memory: once in the kernel memory, and once in the applications memory, before being processed. A function call is also necessary for each received packet to indicate to applications that a new packet has been received.

Poor use of available memory caches. Caches are available to the CPU to reduce the delay to fetch data from memory. The Linux kernel suffers from a lot of cache misses when handling packets, slowing down the process.

Due to these reasons, new solutions have been proposed that rewrite a part of the network stack on the Linux kernel, as well as the NIC driver. These solutions use pre-allocated memory slots to store packet descriptors. They also give applications a direct access to the part of kernel memory where packets are stored. Their design is made very carefully to take full advantage of available caches and to use the memory with the fastest access: all cores of a CPU are not always able to access the same parts of memory at the same speed.

Examples of novel network stacks include the PFQ [BDPGP12b] stack, as well as PF_RING DNA [RDC12] or Netmap [Riz12]. The latter is different from the others because applications do not have to be rewritten to take advantage of the new network stack, while others have custom functions to use. It is also designed to be independent of the NIC driver. These design choices make it capable of handling less traffic than other choices.

Using these custom network stacks, the maximum supported data rate approaches 40 Gb/s. For example, a software router is able to forward packets on commodity hardware at 39 Gb/s [HJPM10].

With these complex network stacks, a lot of software is implied to transmit packets to an application, first in the NIC, and then in the kernel. And to avoid sending too many interruptions, packets may be stored in buffers before being processed. This has an impact on the **reliability** of the platform. For example it is impossible to assign precise timestamps to packets [MdRR+12]. It is also very difficult to guarantee precisely the maximum speed rate of the platform because resources on the CPU are shared with all programs running on the computer, which may affect the packet processing. This article [BDP10] shows for example that the data rate of software traffic generators is unreliable. The main cause is the interference between different processes working in the generator. Multiplie processes work on traffic generation tasks like time-stamping or logging, and other processes are not directly related to traffic generation but manage the user interface for example. All these processes compete for CPU ressources, and switches between processes can slow down the traffic generation unexpectedly.

The **future scalability** of the platform depends on the future commercially available NICs. As the interest for this technology is high, NICs with higher data rates will become available. But exploiting it will also make it necessary to increase the communication speed between the CPU and the NIC, the memory access speed, the processing power, and probably to adapt again the network stack. This is an important amount of work implying different specialized teams, which will have to cooperate to make speed improvements a reality.

2.2.2 CPU computation

Once received data is transmitted to the application, the actual algorithm processing can start. Depending on the application, it can prove very heavy. The difficulty is that CPUs are not specifically designed for traffic processing. This limits the **computation power**: the main problems are related to memory accesses [HJPM10] because the access latency can be long, and the bandwidth gets

very limited if random accesses are done, which is the case for example for hash table lookups that network applications might do for each received packet.

To still be efficient and reach wire speed, the implementation has to be tailored to the hardware architecture:

- The parallelization is important: to get the best efficiency, all cores of all processors have to be used. Depending on the network stack, at least one core per incoming queue of the NIC is already used for receiving packets. For the remaining cores, independent tasks must be found that can run concurrently, to avoid as much as possible sharing data between threads, which is a costly process.

- Although the communication should be minimal, reading packets data in memory is an essential step for all applications. Modern computers use a Non Uniform Memory Access (NUMA) architecture: each processor has an area in RAM with low access delays, it can access other RAM areas but with higher delays. So the location of the memory for each NIC queue is chosen to minimize the writing delay from the corresponding CPU core. Each thread has to be manually assigned to a specific core to optimize all communication delays, and the interrupts sent by the NIC for each queue must be configured to be sent to the corresponding core.

- If the algorithm is very heavy, even the parts that do not directly receive packets will have to be optimized to work on the chosen CPU to keep up with the received data speed. For example, copying data in memory should be avoided, and dynamic allocation of memory replaced by pre-allocated memory.

The most adapted development language for this kind of low-level and highly optimized applications is C++ because it benefits both from the low-level control of C, and from the maintainability and flexibility of object-oriented programming. The **development time** of the application is increased by the need for optimization. For example, this article [SdRRG+12] presents the complex development and optimization process of a 10 Gb/s statistical classification algorithm on commodity hardware.

The interest of pure software development is that it is widely used and benefits from the best **platform openness**: the full Linux kernel is open-source and all major high-speed network stacks like PFQ or PF_RING DNA are open-source too. Development in C works on all platforms, although the different network stacks are not compatible. Migrating to a different platform would require to rewrite the code for receiving packets, and probably to work again on the optimization for the new platform.

The **flexibility** of CPU development is very good as CPUs are designed to be able to handle all kinds of tasks with good performance.

In terms of **security**, the use of the Linux kernel is both an advantage and a drawback [Cow03]: many people work on making it secure, but flaws are also researched actively by attackers. The complexity of the full Linux operating system and applications also gives more opportunities to find flaws.

The **update simplicity** of software development is very good, as processes for online software updates are well-known and used everywhere. Many full-featured solutions are freely available [MBdC+06].

2.2.3 GPU computation

When massive computations are required, GPUs are more and more suggested as a way to overcome the limitations of CPUs. The main interest of GPUs is that they provide massive parallelism with hundreds of cores and a big high-bandwidth memory [HJPM10]. They also have a way to hide memory accesses by having more threads than cores, and running the ones that are not waiting for a memory access first.

This architecture brings more **computation power** for heavy computing on large data. But using the GPU also adds some constraints, that must be taken into account to leverage the computation power:

- The GPU may not replace the CPU as no direct connection can be made between the GPU and the NIC. So the CPU still handles the transmission of packets between the application and the NIC. The GPU is only used to offload some of the heaviest processing from the CPU.
- The communication speed between the CPU and the GPU is limited. The connection if often done using a PCI express (PCIe) link with effective data transfer rates up to 5.6 GB/s from host to device and up to 3.4 GB/s from device to host [HJPM10] (PCIe 2.0 x16). This makes sending all data received from the network to the GPU usually impossible.
- Launching new tasks on the GPU implies significant overhead. For this reason, handling packets one after the other is inefficient. Batching tasks for multiple packets often improves performance, although it also increases the average response time of the application.

Examples of applications using the GPU include a software router with accelerated Internet Protocol version 6 (IPv6) routing lookups supporting up to 39 Gb/s [HJPM10], a traffic classifier with accelerated DPI supporting up to 6 Gb/s [SGV+10] and an intrusion detection system with accelerated pattern matching supporting up to 5.2 Gb/s [VPI11]. These examples all focus on the importance of choosing the part of the processing that will be offloaded to the CPU wisely: it must benefit from massive parallelism, not require too much data transfers, and be easy to batch in big enough tasks.

The **flexibility** of GPU development is actually as good as for CPU development because the CPU is still available for tasks where the GPU is not adapted. But if the main tasks are not adapted, then the GPU probably should not be used at all.

The **development time** is more important than using only a CPU due to the more complex design. The development itself is more complicated and less developers are expert at it. Two main languages exist for scientific development on GPU: CUDA and OpenCL [DWL+12]. CUDA is developed by Nvidia and works

only with Nvidia GPUs, while OpenCL is an open standard designed to work with all GPUs and other platforms. CUDA offers more high-level tools and is more advanced than OpenCL. Efforts exist to make these languages simpler [ULBH08].

The choice of language also affects the **platform openness**. While code in CUDA can be used only on Nvidia GPUs, code in OpenCL can be used on many GPUs, as well as on other kinds of devices, like FPGAs.

The **reliability** and the **security** of the platform is about the same with or without using a GPU, as it is just an offloading device that is not directly in relation with the network interfaces. The **update simplicity** is a bit smaller as online update of GPU software is not as common, but deploying CUDA or OpenCL executables remains simple.

The hardware cost can vary widely depending on the required power. Nvidia Tesla GPUs, which are made to run scientific applications, cost between $1 000 and $3 000 [Ama13], but a full system (NIC, CPU and GPU) may be obtained at only $2 739 [VPI11].

2.3 Network processors

Instead of using commodity hardware, with CPUs that are not designed for traffic processing, processing units tailored to this kind of applications can be used. They are called NPUs. This section will describe the principles and use cases of these specialized processors.

2.3.1 Principles

The main advantage brought by NPUs is a direct communication between the processing units and the network interfaces. As the processors are designed to work with these interfaces, they transmit data at full rate to and from the application, without using the computation units available for the application. Another advantage is that network processors come with some common network functions implemented in hardware, which accelerates their execution.

Network processors form a wide variety with very different designs depending on the requirements: some are specialized for very specific tasks, some are more flexible and generic. But they are typically made of [Hep03]:

Network interfaces which define the supported data rate of the processor. They work just like a NIC except that they are connected directly with other components, without the need of an important software network stack to manage them.

Network co-processors are specialized pieces of hardware that perform a single task. This way, the most common tasks for network applications are directly handled in hardware:

Lookup engines are used for routing, by selecting a route from an IP address for example.

Queue management is often useful to manage congested links: packets are stored and sent as soon as possible, with the ability to give some packets higher priorities than the others.

CRC calculation is required by many protocols to detect transmission errors. They typically have to be computed when receiving and when sending packets.

Encryption functions are useful for security to prevent anyone reading the packet from understanding its content, or to sign a packet and guarantee its origin.

Memory is an important component and may be used to store received data, routing tables, etc.

Programmable processors can be used in the same way as small CPUs, except that they have a direct access to all other components of the network processor. They are usually programmed in C with some special functions available to use the co-processors. They are designed specifically for traffic processing, with an adapted architecture.

This architecture is at the origin of the increased **processing power** of NPUs. Processors are helped by a direct access to network interfaces and very efficient co-processors. This benchmark [MMSH01] compares the performance of an Intel NPU and a CPU and shows that the NPU performs better for all traffic processing tasks. The gains are impressive, but only for applications that do need the functions provided by co-processors. If the bottleneck of the application is in a function that is not common in networks, gains will be much less impressive.

The direct access to interfaces also helps in terms of **reliability**: network processors are guaranteed to work at line rate because of their much simpler network stack.

2.3.2 Development platforms

Development platforms for NPUs are very different from each other. For example, Marvell has [Mar13] network processor chips that have to be integrated on a board with network interfaces, memory, and optionally additional co-processors. The **supported data rate** can go up to 160 Gb/s with multiple 10 Gb/s interfaces. Requesting a quote for the Marvell X11-D240T, which supports up to 40 Gb/s, we were given a price of $250. But the actual **hardware cost** is much higher because the price is only for the chip: an integrated board with network interfaces, memory, and optional co-processors will cost much more than the chip itself.

A very popular development platform is based on the Intel network processors [Int07] but they stopped producing these processors. The reason may be that the market was smaller than expected, as explained by this article [Sch06]. Indeed, NPUs have very specific use cases, so they are confined to small markets. In terms of platform, Intel NPUs use the same principles as the Marvell processors.

EZChip Technologies also sells network processors, which can handle up to 240 Gb/s of traffic [Tec13]. These processors are offered standalone or in full-featured evaluation boxes. The announced speed is impressive, but this is the maximum speed when no application is implemented on the NPU: packets are simply forwarded. The impact of the application of actual supported speed is important.

The particularity of these three platforms is that they are fully incompatible with each other. That means that code developed for one platform would have to be rewritten in big parts to work on a new platform: the framework is different, the available hardware-accelerated functions work differently and the control architecture is different too. This is bad for the **platform openness**. As the development is very low-level, portability is very hard. The C language is used by the three platforms, but assembly development is even advertised for more optimized results on the EZChip development platform.

The complexity and specificity of each platform also increases the **development time**, and finding developers who know the selected platform will prove very difficult. But efforts exist to simplify the development and portability: the concept of virtual NPU has been suggested to develop an universal NPU development framework [BR09], as well as a common architecture for all NPUs [VM04]. These initiatives do not seem to be followed by manufacturers yet. An example development [LHCK04] shows the complexity issue: four algorithms were tested but they had time to implement only one on NPU.

The **update simplicity** depends on the platform, but building executables for the included processors should be easy. The deployment can then be done remotely if a programming interface is accessible.

In terms of **future scalability**, the potential is very high. Alcatel-Lucent claims to use a proprietary network processor in its products able to handle 400 Gb/s [AL11]. The problem is that once a platform is chosen, changing is very costly, so improvements depend fully on the will of the manufacturer. This is concerning as some major manufacturers like Intel have discontinued their network processors family [Int07].

The **security** of each platform depends on the manufacturer. The network stack is simple, which means that software running on the platform is limited and controlled either by the manufacturer or by the application developer. This limits security risks.

2.3.3 Use cases

Depending on the application, using network processors will bring different performance improvements. If the application relies massively on common network functions like lookup engines, NPUs will be efficient accelerators because these functions are implemented in hardware [MMSH01]. But if the bottleneck is due to a more original function, that is not available as a co-processor, improvements will be less impressive.

As vendors offer solutions with different architectures and co-processors, the best platform choice depends on the requirements of the developed application.

As changing during development would be very costly, making a wise choice from the start is essential. This also limits the **flexibility** of the platform, as some applications will not be adapted because the speed bottleneck is an uncommon task in networks.

An example use case is an application for flow management [QXH+07]. It receives packets and builds information about UDP and TCP flows. It is also capable of remembering the state of each flow. They use the Intel IXP2850 network processor. The process of gathering flows relies heavily on hash tables, which is one of the common functions implemented on NPUs. To get better performance, they use the CRC hash function, which is already implemented on the NPU. This way, they support the line rate of 10 Gb/s. This kind of applications is very adapted to an NPU implementation.

Another possible application is traffic classification using DPI [LKL12]. They use a network processor with 4x10 Gb/s interfaces. This example shows the flexibility of network processors: modular software development is used to be able to change on the fly parts of the code executed on the NPU.

2.4 FPGAs

Field-Programmable Gate Arrays (FPGAs) are special integrated circuits that can be configured as many times as necessary at a very low level. Their big difference with all previous solutions is that they are not processors and are not made to be programmed. They work at a lower level: it is actually possible to implement CPUs, GPUs or NPUs using an FPGA. Systems on chip exist that associate FPGAs and CPUs, and if programmability is needed, a CPU can always be implemented on the FPGA, but this is not the main interest of FPGAs. This section will describe how these circuits work, how they can be used for network applications, and their interests.

2.4.1 Composition of an FPGA

An FPGA is a reconfigurable integrated circuit. It basically provides three types of very low-level elements [RAMV07]:

Inputs and outputs are just ports capable of receiving and sending binary data outside the chip to communicate with other elements.

Look-Up Tables (LUTs) are components that can realize any combinatorial logic function: they receive a certain number of words as input, and set their output bits to a combination depending only on the input bits. The LUTs can be configured so that any output corresponds to any input. This way, a LUT with four input bits and three output bits can be used for example to realize an adder of two words of two bits: the two first input bits are used for the first word, the two others for the second word, and the output is the result of the addition of the two words. Other basic functions like a logical or, a logical and or a multiplexer (to implement conditions) can be realized.

Registers are small local temporary memory elements. They take as input a word, an enable signal and the global clock signal of the circuit. When the enable signal is active, they change their output at each clock cycle to the input word. When the enable signal is inactive, the output keeps its current value. Registers allow to synchronize the values in the circuit on the global clock. They also allow to keep temporary values in memory.

These three types of elements are present in very important quantities in an FPGA. Configuring an FPGA simply consists in drawing wires between these elements, and configuring the function realized by each LUT. All complicated operations on bits can be realized this way. Of course, building complex circuits this way would take a very long time. This is why development tools exist that support all basic operations: arithmetic operations, boolean operations, conditions... The wanted circuit is realized at a higher level, and an automatic tool decides how to use the LUTs and how to do the wiring.

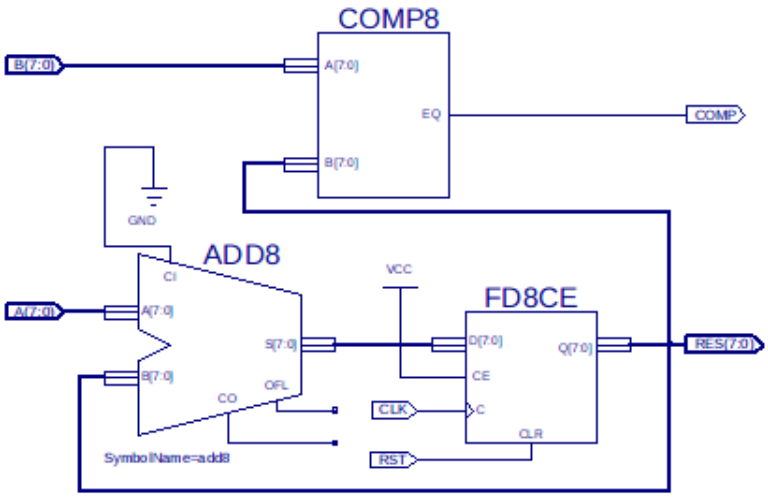


Figure 2.1: Sample circuit on an FPGA realized with Xilinx ISE

Figure 2.1 is an example of circuit that can be configured on an FPGA. It has two inputs A and B. Output RES is the result of the accumulation of the A values at each clock cycle. Output COMP is the result of the comparison of RES with B. Here LUTs will be used to implement the addition and the comparison. Registers will be used to store the result of the addition at each clock cycle, so as to reuse it during the next clock cycle to compute the accumulation.

Although LUTs and registers are the main components of an FPGA, some other elements are often present to simplify certain operations:

Memory is often available on the chip to store bigger quantities of data than the registers can contain. This memory is usually small but can be read or written in only one clock cycle, which makes it very easy to use.

Digital Signal Processing (DSP) units are also available to realize some complex arithmetic operations. They are often used as multipliers. They are faster and use less space than the same function implemented in LUTs.

Transceivers are special inputs and outputs made to connect the FPGA to links that work at a higher frequency than the FPGA can directly handle. They are essentially serial-to-parallel converters in input and parallel-to-serial converters in output.

The **computation power** provided by FPGAs comes from their inherent massive level of parallelism. Processors work by processing instructions which depend on each other, so the instructions are executed one after the other. On an FPGA, all elements can work concurrently. If the design is well done, no part of the circuit will be idle at any time during processing. For example on Figure 2.1, both the adder and the comparator perform one operation at each clock cycle. On a larger design, thousands of operations happen at each clock cycle. Here [GNVV04] is an analysis of the speedup factor of an FPGA over a CPU.

An FPGA also provides a very good **flexibility**. All kinds of applications can be developed on it. Compared to CPUs, it is not originally designed for floating-point computations. It can still be done [Und04], but it increases the complexity and requires a lot of resources on the chip. This is why most applications are converted to fixed-point computations when ported to FPGA. If a variable in an application requires a very good accuracy on small numbers and a big range of values, or if a variable may not be bounded, the conversion to fixed-point computations will make the results inaccurate. In this situation, using an FPGA is probably not the best solution.

The two main manufacturers of FPGAs are Xilinx and Altera. They are used for very diverse applications [RAMV07] like medical imaging, signal processing, error correction codes, etc.

2.4.2 Boards for traffic monitoring

An FPGA is just a computation unit. It receives binary data, processes it, and generates binary data. It has a large number of input and output wires. This is why it is usually integrated on a board, which connects the FPGA to other electronic components.

Some boards are specialized for network applications. They mainly connect the FPGA to network interfaces and memories. Most of these boards are designed to be embedded into a computer, and are able to communicate with the CPU through a PCIe port. Among these boards are:

The NetFPGA 1G [LMW+07] embeds a Xilinx Virtex-II Pro 50 FPGA, 4x1 Gb/s Ethernet interfaces, 4.5 MB of SRAM and 64 MB of DRAM.

The NetFPGA 10G [Net12] embeds a Xilinx Virtex 5 TX240T FPGA, 4x10 Gb/s Ethernet interfaces, 27 MB of SRAM and 288 MB of DRAM.

The Combo 20G [IT13] embeds a Xilinx Virtex 5 LX155T FPGA, 2x10 Gb/s Ethernet interfaces and 144 MB of SRAM.

The NetFPGA 10G represents a **hardware cost** of $1 675 for academics. The only other hardware required to use it is a normal computer that will communicate with it and be able to reconfigure it. The most obvious difference between these three boards is the speed of their interfaces, but the FPGA model differs too. Different models have different numbers of logic units (LUTs, registers, inputs and outputs, memory and DSP units). The most powerful FPGA is on the NetFPGA 10G, and the least powerful on the NetFPGA 1G. The available on-board memory varies as well. Other models of FPGA exist that are much bigger, but they have not been chosen for these boards to keep the cost low.

The **supported data rate** of the boards only depends on the speed of their interface. Network interfaces are connected to the FPGA in a very direct way. For example on the NetFPGA 10G, each 10 Gb/s optical link is connected to an enhanced Small Form-factor Pluggable (SFP+), which converts the optical signal into an electrical signal. This signal is then connected to an electronic dispertion compensation equalizer, which cleans the signal, and to four RocketIO GTX transceivers, which are special transceivers directly integrated to the Xilinx Virtex 5 FPGA. This way, the signal is seen inside the FPGA as two parallel buses that work on the FPGA frequency and support the 10 Gb/s speed. The actual speed of the bus is even higher than 10 Gb/s to allow to add internal headers to each packet to transmit information between different entities implemented on the FPGA. There is one bus receiving data, and one bus sending data. This article [BEM+10] describes more deeply the architecture of the NetFPGA 10G board. All the application has to do to support the maximum speed is to not slow down the buses. This also affects the **reliability** of the platforms: the boards are designed to support the maximum data rate of their interfaces with the smallest packets. The application developers control all the factors that may slow down the computing as they decide how to use each electronic component.

To guarantee the **future scalability** of applications developed on these boards, different solutions are possible:

- Use FPGA boards with faster interfaces: a Combo 100G board is for example in development [Pus12]. It will have one Ethernet interface at 100 Gb/s.

- Reuse the FPGA development to design an Application-Specific Integrated Circuit (ASIC). These specialized circuits are made of basic logic and arithmetic gates to realize a specific function. Contrary to FPGAs, they cannot be reconfigured after manufacturing. This means that all the reconfiguration logic that takes a lot of space on an FPGA can be removed, and that basic logic gates can be optimized. This way, ASICs reach higher speeds and are more compact, which would help support higher data rates. A design for an FPGA is usually used as base to design an ASIC. Specialized circuits cost more to design, but if a lot of units are sold, each unit will cost less than an FPGA.

2.4.3 Development principles

Development on FPGA is usually based on one of two languages: VHDL or Verilog. Both are hardware description languages. They look similar to classical programming languages, but do not work the same way. The main difference is that lines written one after the other do not execute consecutively but concurrently. A file usually describes one entity, with input and output wires, and the description of the processing made by this entity. Entities can include other entities as components by connecting to their inputs and outputs.

Full-featured development suites are provided both by Xilinx and Altera. Each suite works only with FPGAs from the same manufacturer, but development can always be done using indifferently VHDL or Verilog. So code developed for one platform can be used on the other. The important tools for development on FPGA are:

A simulator, which is used to check the proper behavior of an entity described in VHDL or Verilog. All there is to do is to write a testbench in VHDL or Verilog, which will manipulate the input values of the entity during the simulation time. The graphical simulation shows everything that happens on each wire inside the entity. It is also possible to automate the verification by checking the output values against predicted ones. This tool is essential to save time and ensure that no obvious development mistakes were made before testing the entity on the FPGA.

A synthesizer, which transforms an entity written in VHDL or Verilog into a bitfile that can be used to configure an FPGA. This process is made in many steps and can take a long time. The code is first transformed into a list of basic operations that are available on the FPGA. Operations are then mapped on the actual layout of the FPGA, and the tool tries to route all connecting signals on the chip. This is a heavy optimization problem: each bit manipulated by the entity is a wire, and if any wire between two registers is too long, the global clock frequency of the FPGA will have to be lowered, which will reduce the computation speed.

Once the bitfile is configured on the FPGA, its functionality can be tested. But it is a very slow and complicated process. The main problem is that is is difficult to access to the internal values of wires inside the FPGA. This is why the simulation must be done very thoroughly to avoid having to debug hardware.

The **development time** on FPGA is longer than on processors because it is very low-level. Simple operations can take long to implement. Software developers will also need time to get used to the different development paradigms on FPGA. The debugging process can also be very long, especially if the bug is not visible during simulation, because the synthesis operation takes very long, and access to internal wire values of the FPGA is complicated. A comparison of the development of the same application on FPGA and GPU [CLS+08] shows that FPGA development is longer.

To support the specific features offered, NetFPGA and Combo boards come with a development framework. The NetFPGA framework is open-source and

free. Some design choices are different between the NetFPGA 1G and the NetFPGA 10G [ASGM13]. The Combo framework is closed, it is called NetCOPE. It is compatible with all current Combo boards and recently with the NetFPGA 10G [KKZ+11]. Each framework contains the description of the global entity that can be used on the NetFPGA, with the configuration of the connections to all components on the board. The global entity is made to connect to the network interfaces and manage incoming and outgoing traffic. An application developer just has to design an entity, which will connect to buses to communicate with the network interfaces, and respect the specified bus protocol. A simulation environment is also provided to simulate the developed entity in a context as close as possible to the actual board.

The **platform openness** is better for NetFPGA because the framework is fully open-source, and open-source projects using the platform are encouraged and listed on their website. But the NetCOPE platform is compatible with more boards. It can be noted too that thanks to the use of VHDL or Verilog on both platforms, and the inherent modularity of hardware description code, migrating from one platform to another is not very difficult. It essentially implies to adapt the bus interfaces in input and output. The existence of an adaptation of the NetCOPE platform for the NetFPGA 10G is an evidence of the similarity of their design [KKZ+11].

The **update simplicity** of FPGAs is not very good. As the process to generate the bitfile (which is the equivalent of an executable for an FPGA) is very long, even small changes require some time. And as development is more complex, the validation phase should be more thorough. The workflow is described more precisely here [RAMV07]. Reconfiguring an FPGA can be made remotely if its reconfiguration interface is connected to a computer on the network. But FPGA implementations can be made much more flexible as it is even possible to configure a programmable CPU on an FPGA, enabling simple software updates. For example, this article [LSS+09] implements a specialized CPU made for forwarding traffic on a NetFPGA 1G. This is a custom NPU with hardware accelerators specifically designed for the application.

Ensuring the **security** of an application on FPGA is simpler than on a CPU because less external code runs on it. The code that does not belong to the application developers belongs to the used framework. It can be open-source code, but there are also closed-source blocks, called Intellectual Properties (IPs). For these parts, developers have to trust the providers.

2.5 Conclusion

Different development platforms are adapted for network applications. Depending on the requirements, the best platform to choose may be different. The advantages and drawbacks of each platform are summarized in Tables 2.1 and 2.2.

If the main requirement of the application is to be flexible, using commodity hardware seems to be the best solution because development and updates are easier. Depending on the needs in parallelization, using a GPU will be necessary or not.

If the goal is to have the highest data rate and computation power, a choice has to be made between NPUs and FPGAs. NPUs are very good for common network tasks, and may be cheap if hundreds of thousands of units are sold. FPGAs are more flexible and the developed code is more easily portable. If more computation power is required and hundreds and thousands of units are to be sold, ASICs should be considered, as development on FPGA is a good base to design an ASIC.

In next chapters, three applications will be developped. One on commodity hardware and the two others on an FPGA. The results of these developments will shed more light on some aspects of these platforms, especially:

- the compared **computation power** available on CPU and on FPGA, and how to exploit it best,
- the **update simplicity** provided by CPUs and the ways to get some flexibilty on FPGAs too,
- the **development time** on each platform, and the possibilities to reduce it on FPGA,
- the **future scalability** perspectives provided by each platform to the applications.

	CPU/NIC	CPU/NIC/GPU
Supported data rate	Interfaces up to 4×10 Gb/s [Int13]. Supporting about 40 Gb/s requires novel network stacks and software highly customized to hardware [HJPM10].	
Computation power	CPUs design is not the most adapted for packet processing [HJPM10] (memory access, parallelism level)	GPUs provide a higher level of parallelism and processing power. The CPU/GPU communication bandwidth may become a bottleneck [HJPM10, NI10].
Flexibility	All algorithms can be implemented	GPUs are not adapted if computations require a lot of data communication [AMY09]
Reliability	Accuracy of timestamps and supported data rates is low [MdRR+12, BDP10]	
Security	More complicated software means more risks of security flaws, use of the open-source Linux kernel [Cow03]	
Platform openness	Full open-source network stacks exist (e.g., PFQ [BDPGP12b], PF_RING DNA [RDC12], HPCAP [MdRR+12], Netmap [Riz12]) but are not compatible with each other.	For GPU programming, OpenCL is an open standard, CUDA is specific to Nvidia [DWL+12]
Development time	Pure software development, but great care must be taken to customize the architecture to the underlying hardware [SdRRG+12]	GPU programming is more complicated than CPU [ULBH08], GPU/CPU communication requires careful design [HJPM10]
Update simplicity	Development on CPUs and GPUs is based on C [DWL+12, HJPM10]. After compilation, the executable can be deployed remotely.	
Future scalability	Current implementations reach around 40 Gb/s [HJPM10]. Supporting higher data speeds would require better hardware, and an adaptation of the software part to eliminate the numerous limiting factors (buses speed, memory speed, processing power)	
Hardware cost	Intel 40 Gb/s NIC: $1 100 [Int13]	Nvidia Tesla GPU from $1 000 to $3 000 [Ama13]

Table 2.1: Advantages and drawbacks of different development platforms (1)

	NPU	FPGA
Supported data rate	1 Gb/s and 10 Gb/s up to 160 Gb/s at Marvell [Mar13]	Interfaces up to 4 × 10 Gb/s [Net12]. Full support guaranteed by design [BEM+10].
Computation power	High parallelism level and specialized accelerated functions (hashing...) [Hep03, MMSH01]	FPGAs provide massive parallelism and low communication latencies [GNVV04]
Flexibility	Possibilities depend on each model. Very common tasks beneficiate from the most acceleration	FPGAs may not be adapted if numerous floating-point computations are essential [Und04]
Reliability	Designed to work at full rate [Hep03]	Designed to work at full rate [BEM+10]
Security	Risk of security flaws in the APIs provided by manufacturers like Marvell [Mar13]	Risk of security flaws in the IPs provided by manufacturers like NetFPGA [BEM+10]
Platform openness	Each vendor has its proprietary framework, although efforts exist to unify them [BR09, VM04]	Verilog and VHDL are languages that work for all FPGAs, packet handling frameworks are similar and conversion is easy [KKZ+11]
Development time	Development complexity is high [LHCK04] and dependent on each platform.	Development on FPGA is the most complex [CLS+08, RAMV07]
Update simplicity	Development on NPUs is mostly based on C [Mar13], remote deployment is possible only if it was included in the board design.	To reconfigure an FPGA, a bitfile is required. The process to get it is called a synthesis, it can be very long (1 day) [RAMV07]. But it can be made very flexible (programmable) if designed so.
Future scalability	Manufacturers claim supporting up to 400 Gb/s using proprietary (unavailable to the public) network processors [AL11]. Scaling requires changing the processor.	The current supported speed is 40 Gb/s [Net12]. A 100 Gb/s board is in development [Pus12]. Scaling requires essentially adapting to a new communication bus width. An FPGA implementation can also be used to design a more powerful ASIC.
Hardware cost	Network processors are often sold alone, they must be integrated on a custom board. Marvell X11-D240T processor costs $250 (negotiable)	NetFPGA 10G board academic price: $1 675 [Net12]

Table 2.2: Advantages and drawbacks of different development platforms (2)

Chapter 3

Software monitoring applied to security

The most straightforward way to implement traffic monitoring is by developing software on commodity hardware. This is a solution we studied in the frame of an European project called DEMONS. One of the goals of this project is to provide a flexible framework for traffic monitoring applications at high data rates. Our role in DEMONS is to test this framework by developing a test application. As DEMONS is focused on security, the chosen use case is to detect a specific kind of attacks called Distributed Denial of Service (DDoS). This choice is especially due to previous work [SVG10] made at Télécom Bretagne on DDoS detection for a French Research project called Overlay networks Security: Characterization, Analysis and Recovery (OSCAR).

DDoS attacks consist in sending traffic from multiple locations in order to prevent a target from serving normal requests. These attacks are more and more visible in the news [Rya13] because big companies and even states are attacked. They are simpler to stop early in the network than at the level of the target, because attack packets all converge to the target from different locations. This is why network providers should be equipped of DDoS detection and mitigation systems.

We will first describe the state of the art about DDoS detection and software monitoring platforms. We will then explain our focus on detecting a specific kind of attack. The flexible and scalable network monitoring framework provided by DEMONS will then be described. Finally we will detail the implementation of our application in this framework and evaluate the results in terms of scalability.

3.1 State of the art on DDoS detection implementation

In this section, we study parts of the literature on traffic monitoring that are interesting to implement a DDoS detection application in a flexible and scalable way. As we are interested in evaluating the monitoring platform provided by the DEMONS project, we first study different existing monitoring platforms, as well

as their advantages and drawbacks. Then we focus on the DDoS detection use case, first listing different kinds of DDoS attacks, and then describing different ways to detect these attacks.

3.1.1 Monitoring platforms

It is possible to develop a DDoS detection system as a standalone application. But to improve the flexibility and hopefully reduce the development time, existing monitoring platforms can be interesting. For the user, the monitoring platform defines the ease to install the system on the network, the maximum data speed that may be supported, and the way the system is used and maintained. It is also important for the developers because it can make the implementation easier or harder.

CoMo [Ian06] is an example of network monitoring platform. Its principle is to work on different time scales depending on the task:

- The capture, that is to say the extraction of certain simple features in the flow of monitored packets, is made in real-time. This is the most time-constrained operation.
- The export, which consists in storing the extracted features into a database, is periodic.
- The advanced analysis on received data is made at user request.

The point of this differentiation is to give the most resources to the capture, so that it can process a large amount of traffic. Application Programming Interfaces (APIs) are available to control each task, so that developers can customize their behaviour. This means that developers write code that has to run in real-time, but the CoMo API provides the capture and export primitives, as well as the user interface.

ProgMe [YCM11] is another monitoring platform that is more strict to developers than CoMo. The goal is to make sure that developers will not hinder the performance of the capture mechanism. ProgMe defines the concept of flowsets, sets of packets with common properties (like the same source IP or the same source and destination TCP ports). The real-time capture engine updates configured counters when it receives packets that belong to the corresponding flowset. Flowsets can be programmed dynamically. That means that developers do not write code that is executed in real time, but only code that configures the real-time capture engine. This is less flexible than CoMo, but easier to scale to higher bit rates.

It is also possible to choose a different direction and give as much flexibility as possible to developers, so that they are responsible of the scalability of the end result. This idea can be found in the modular router Click [KMC+00]. This is not a monitoring platform but it is also designed to handle packets. Here developers can add elements of code that are inserted in the datapath: they act directly on packets. It is even more flexible than the principle of CoMo because there are no fixed tasks: developers can add as many elements as they want to modify

the behaviour of the router. We will see that this system can be applied to a monitoring platform.

As discussed in the previous chapter, for a software monitoring platform to support high bit rates, the network stack is very important. The Voice over IP (VoIP) monitoring system RTC-Mon [FHD+09] is an example of platform using a fast network stack, PF_RING [RDC12].

In the frame of the DEMONS project, a network monitoring platform called BlockMon has been developed. Its specificities are described in Section 3.3. We will now focus on the use case made to test the BlockMon framework: a DDoS detection algorithm.

3.1.2 DDoS attacks

Attack techniques

DDoS attacks are more and more widespread because they are both simple and efficient. They consist in using a large number of computers to attack a single target, so that it stops working normally. Results of these attacks are visible in the news [Rya13]: big companies under attack are not able to provide their online services for long periods, and the global traffic on the Internet can even significantly increase [Mat13].

The usual goal of these attacks is to disrupt the normal operation of the target. This is used for example by the "Anonymous" group to protest against companies for ideological reasons. The website of Paypal for instance was under attack in 2010 [New13] because Paypal blocked accounts financing WikiLeaks. But DDoS attacks can also be used as part of a more sophisticated attack to get access to restricted resources. Once a server is out of order because of a DDoS attack, it is easier to replace it by a fake server that other servers will trust, and that is controlled by the attacker. For example, most local networks are equiped with a DHCP server that provides the address of the router to all machines arriving on the network. Setting up a fake DHCP server to provide a wrong router address is easy, but the selection of the real or fake DHCP server by new machines arriving on the network will be random. A DDoS attack on the real DHCP server can prevent it from answering and make sure that all computers use the fake DHCP.

Many different kinds of DDoS attacks exist, but the basic principle is mostly the same: it relies on sending multiple times a request that is:

- easy to generate, so that it can be sent at a high rate to the victim;
- difficult to handle for the destination server, so that a low request rate is enough to use all resources on the destination server;
- difficult to differentiate from normal traffic, so that the attack cannot be filtered easily.

Classical DDoS attack techniques use inherent flaws of various layers of the network stack [JP12]:

At the network layer:

The teardrop attack targets the reassembly mechanism of the IP protocol. Invalid IP segments are sent to the target, giving it a lot of useless work to try and reassemble the segments.

The Internet Control Message Protocol (ICMP) attack consists in sending ICMP requests, called "ping" to the target, so that it replies with ICMP echo packets. The goal is to create network congestion and use the target's resources.

At the transport layer:

UDP flooding consists in sending many UDP packets to random ports of the victim, to force it to reply with ICMP network unreachable packets.

TCP SYN flooding works by sending many TCP SYN packets to the victim. The TCP protocol is flow-based, which means that a state has to be maintained in memory on the two machines implied in a TCP connection. The TCP SYN packet is designed to start a connection, so it forces the receiving machine to allocate memory to manage the new connection, and to send acknowledgement messages. By sending many TCP SYN packets to a target, its resources are intensively used.

TCP reset is a bit different from other attacks because it does not consist in sending a massive amount of packets. It simply consists in sending a TCP reset message to the victim, making it look as if it belongs to an existing connection, with an existing source address and port, and existing destination port. This way, the legitimate connection is stopped and the victim cannot provide the expected service. This is a small scale attack that can be useful to stop one specific connection.

At the application layer:

The DNS request attack is interesting to attackers because it contains an amplification mechanism: many DNS requests are sent to domain name servers with the source IP address set as the IP of the victim. Domain name servers reply to each request with data about the requested domain. But as the source IP address of the requests was forged, the replies are sent to the victim. The replies are much bigger than the requests, so attackers only have to send small requests, and the victim is flooded by big messages.

The mail bomb works by sending many email messages to the address of a victim. This will fill the disk of the victim, making it unable to provide its normal services.

Another important aspect for attackers is the need to mask the sources of the attack, to make it more difficult to identify the attacker. The best way to do this is to use reflectors, that is to say open servers on the Internet that easily reply to large amounts of requests. Attackers send requests to this server with a forged source IP address, set to the address of the target, so that the server answers to

the target instead of answering to the attacker. This is the principle behind the DNS request attack: the reflectors are then domain name servers.

With the same principle, a variant of the TCP SYN flooding attack is also used: instead of sending SYN packets directly to the victim, they are sent to an open server, with the source address set to the address of the victim. This way the server replies with TCP SYN ACK packets to the victim. Contrary to the previous attack, this variant contains no amplification mechanism: SYN ACK messages are the same size as SYN messages. But the source of the attack is masked.

These two kinds of attacks using reflectors (DNS requests and TCP SYN flooding) have been involved in a recent attack against Spamhaus [Mat13].

Generating the traffic

The last remaining problem is that most attackers do not have the resources required to send large DDoS attacks, and they do not want to send these attacks using their own machines as this would make them easy to identify. This is where botnets come into play.

Botnets are sets of remotely-controlled machines used to perform a common task. Although legal botnets exist, attackers use illegal botnets. The difference is that they are usually made of machines from normal Internet users, who have no idea that their machines are parts of a botnet. Viruses are used to take control of these machines: once the virus has gained access to the machine, it installs a backdoor that enables the pirate to send orders to the machine. These orders are usually simple: "send TCP SYN packets to this address", "send this email to these addresses" or "send random DNS requests to this server".

Actions from the botnet seem to come from innocent Internet users who are not even aware of what their machines are doing. The only way that remains to identify the pirate is to find the way it controls the botnet. So pirates have made it very difficult to find whom the backdoor communicates with. Some backdoors connect to Internet Relay Chat (IRC) servers on a given channel, and wait for the attacker to send a specific control message [CJM05]. Others use a centralized command server with an address masked using multiple domain names stored in a list by all bots [SK12], and some even use more Peer-to-peer (P2P) architectures where the order is sent only to some bots that transfer it to the others. Mobile botnets running Android are even controlled by Short Message Service (SMS) [NP12].

Botnets are usually not directly used by the pirates who create them. Others pay the pirates to use their botnets. One common use is to send massive amounts of spam messages. Generating traffic for DDoS attacks is also a very frequent use. It is an efficient way to get large resources, while keeping the risk to be identified very low.

As identifying attackers and finding or destroying botnets are very challenging tasks, the best current way to defend against DDoS attacks is to mitigate their effects when they happen. Currently commercial services exist that act as a shield between a server and the attackers. They have farms of servers located all around the world, replying to the same IP address. This way attacks are spread

between all servers that receive fewer packets, and are able to ignore them. This routing method is called anycast. Network providers use a protocol called BGP to communicate between each other where each IP address can be contacted. When using unicast, the same IP address is announced at different locations on the network. Routers choose the nearest location.

This technique dilutes the impact of the attack but does not actually stop it. Only operators have the ability to mitigate attack traffic on their network. This is an interesting service that network operators could provide to their customers: a guaranteed protection against DDoS attacks. To do that, attacks must first be detected. We will now see different ways to detect DDoS attacks, focusing on the very common TCP SYN flooding attacks.

3.1.3 DDoS detection algorithms

Depending on the technique used, DDoS detection can be deployed at different levels:

At the server level, only one machine is protected. All inbound and outbound traffic of the server is watched. This allows very fine-grained monitoring. By knowing the role of the server, a list of expected types of communication can be made, as well as a description of the way each client is expected to behave for each type of communication. If a client behaves in an unexpected way, trying to join unexpected ports or sending an unusual amount of traffic, an alarm can be raised. It signals a malfunction or an attack.

An example of such a system is described in this article [PYR+13]. It is located on a router close to the server to protect. It checks all inbound and outbound packets by maintaining state machines. There is one state machine for each communication (for example an Secure SHell (SSH) session or an HyperText Transfer Protocol (HTTP) file download). If the communication brings the state machine into an unusual state, an alarm is raised.

This technique is very accurate, but it is difficult to manage for network administrators: each legitimate service provided by each server to protect must be listed and precisely analyzed. The task of protecting a server under attack is also very heavy for the router, because it receives all the attack traffic targeted to the server, and it has to maintain state machines about this traffic. So there is an important risk for the router to fall under the DDoS attack, while trying to protect the server. If this happens, the server will become unreachable, and the attack will have succeeded.

At the access network level, a whole set of machines on the same subnetwork can be protected at once. The main interest of staying near the network border is that attacks are easier to detect. Indeed DDoS attacks come from different locations on the Internet. So a router in the core network does not see the whole attack, but only a small part. Other packets use different ways to reach their target. Another advantage of the access network is

that for each protected machine, both inbound and outbound traffic can be watched. As routing in the core network is not symmetric, there is no guarantee that a router will see both inbound and outbound packets.

Techniques made to protect a large number of machines from DDoS attacks monitor the traffic in a more global way than at the server level. This makes these approaches more scalable.

At the core network level, the goal is to detect all transiting attacks, the target does not matter. Although attacks are easier to detect in the access network because they are more focused, they are more dangerous too, and more difficult to handle. So detecting and mitigating them in the core network is very interesting, because the load is distributed among many routers. Detection at the core network level requires less capture points but with higher data rates. The detection probes are more complex to design, but the mitigation is simpler at this level.

As the amount of traffic transiting in the core network is huge, approaches at this level must be highly scalable. As the attack is distributed, some collaboration between core routers located in different places can help detect attacks. It can even be interesting for different operators to share data about attacks they detect. This is called inter-domain collaboration. It can be very powerful because operators could act globally. It is also very challenging because operators are usually unwilling to share data about their network with potential competitors. They will not share their number of customers or the details of their transiting traffic, but it might be possible to let them share aggregated data if they have strong enough incentives. An example of incentive could be the ability to offer a protection service to their customers.

Depending on the level, and on the wanted features, two categories of detection mechanisms can be considered [AR12]. The first mechanism consists in detecting attacks using signatures that are known *a priori*. Past attacks are observed and discriminating features are extracted. These features are called signatures, and when they are seen again in the traffic, an alarm is raised. This technique does not allow to detect new attacks that never happened, but they have a very low false positive rate, because signatures match only a very specific behaviour, known to be an attack.

The second mechanism consists in finding anomalies in the traffic. It learns usual features of the traffic when there is no attack, and detects when the features seem to vary in an unusual way. This technique usually detects both attacks and network malfunctions. It is capable of detecting attacks that were not known before, just because they do not resemble normal traffic. The drawback is that normal changes in the traffic, like a sudden surge in popularity of a server, might trigger alerts [LZL+09].

As attackers are very inventive and try to avoid all detection mechanisms, attacks are not very often repeated the same way, so signature-based mechanisms are not very efficient. This is why the research currently focuses on anomaly detection.

The first requirement to design an anomaly detection algorithm, is to choose the features to watch. These features can be very different. The most obvious is the quantity of traffic to a certain destination. It is used in [SDS+06]. But other features are possible, like for example the covariance of TCP flags in TCP flows. These flags made to open or close a connection, or to acknowledge a received packet, are used in [JY04] to detect TCP SYN flooding attacks. The principle is that the ratio between SYN flags, made to open a connection and FIN flags, made to close it, is different for an attack and for normal traffic. Indeed, attackers send SYN packets but do not care about closing the connection. Another example [YZD08] uses the distribution of the number of packets per period of time in a flow. It assumes that, due to their common generating programs, TCP and UDP flows of a same attack all share a similar distribution of packets. By detecting these similarities, they detect the attack. The goal with this technique is to detect attacks that mimic normal traffic, because generated packets are not as diverse as real traffic, even if the generating program is complex. It is also possible to monitor the mean packet inter-arrival time to a specific destination to detect attacks [SKKP12]. This means that a server that suddenly receives a massive amount of traffic is probably attacked. The drawback is that natural surges in popularity are interpreted as attacks.

For this kind of analysis on the amount of traffic, an algorithm called CUmulative SUM control chart (CUSUM) is widely recognized [TRBK06, AR12, SVG10] to be both efficient and simple. It is lightweight and can detect accurately changes in a time series. It will be detailed in Section 3.2.2.

We will now describe the way we implemented the DDoS detection algorithm for DEMONS.

3.2 Flexible anomaly detection

3.2.1 Problem statement

DEMONS is a European FP7 project. The project is over for some months but its aim was to provide the infrastructure for network operators to cooperate on a global decentralized monitoring platform. The expected benefits of this new infrastructure lie in the ability to detect global attacks or failures faster and to mitigate them cooperatively. To be successfully adopted, the proposed infrastructure has to incorporate any kind of monitoring and mitigation applications network operators may want to use. Challenges for such an infrastructure arise at all levels:

Capture flexibility and speed. Multiple capture points can be deployed on the network. At each capture point, metrics must be updated about the received traffic. The capture point can monitor a link with high data rates, so the monitoring application has to support it. But each monitoring application may need to process received packets differently: filter only some packets, keep counts on some features, search for a specific packet. So capture points have to be programmable by application developers, and the

task should be made as easy as possible. The challenge is technical: the capture point must be both flexible and able to process high data rates.

Mitigation interactivity. The infrastructure has to incorporate mitigation applications. But these applications are very different from monitoring applications because they cannot be fully automated. At each mitigation point, actions must be taken only on orders from network administrators. So an user interface must be provided to network administrators so that they can see alerts in the network and make decisions on how to mitigate them. The challenge is about complex data representation: many sources of alerts have to be aggregated together in a meaningful way.

Efficient intra-domain communication. At the intra-domain level, that is to say inside the same network operator, capture points must be able to communicate with each other. Aggregation points may also be implied to centralize and analyze data. This enables distributed traffic monitoring. As each capture point sees only a part of the traffic, cooperation is the only way to get a global view. This is another technical challenge: capture and aggregation points must communicate without generating too much traffic, and the communication must be easy to implement for developers.

Inter-domain cooperation. At the inter-domain level, different network operators cooperate and exchange information about what is happening on their network. At this level, the challenge is about confidentiality: operators do not want to disclose data about their network to their competitors, so any kind of information that is automatically exchanged must first have been approved manually.

Our contribution to the project is mainly on capture flexibility and speed and on efficient intra-domain communication. To assess the efficiency of the architecture, we developed a scalable distributed TCP SYN flooding detection application in the framework of DEMONS. Other applications developed by partners include an heavy hitter statistics system to detect large flows and a VoIP fraud detection system [dPHB+13]. For the DDoS detection application to be deployable on a real network, some requirements are mandatory:

- Alerts should be generated with IP addresses of both the attacker and the victim. If the attack is distributed, one alert should be generated for each attacker.
- Alerts should be exported in a normalized format. This makes it easier for mitigation applications to react to these alerts.
- Detection should be distributed, that is to say that different capture points should be able to collaborate to detect attacks coming from different parts of the network.
- A legitimate increase in popularity of a server generating a flash crowd should not be detected as an attack.

- The deployment and configuration of the application should be highly flexible. The level of distribution of the application, the resources assigned to each monitoring point to support high data rates, the thresholds at which alerts are sent, should all be easy to configure.

With the development of this test application, we contributed to DEMONS a lot of code that can be reused for other traffic monitoring applications. This code is especially focused on counting packets, detecting changes in series, and communicating alerts on the network in a standardized format. We also tested parts of the code we did not develop, essentially for packet capture and inter-machine communication.

Before describing the framework of DEMONS and the way the example application is integrated into this framework, we will detail the chosen algorithm for DDoS detection.

3.2.2 Algorithm for DDoS detection

The list of existing DDoS attacks described in Section 3.1.2 is long. To develop an example monitoring application, we focus on a very common attack: the TCP SYN flooding attack. It consists in sending multiple TCP packets to the victim, with the SYN flag set. This flag indicates the start of a communication, so the victim has to maintain a state in memory each time it receives one of these packets. This way, the machine is fast over-loaded, and it cannot serve its normal clients anymore.

Although we focus on one kind of DDoS attack, most attacks use the same principle: send a large number of packets to a victim from multiple attackers. So the principles we will use to filter packets, count them, and analyze the counts in a scalable way can all be reused for other anomaly detection applications.

The monitoring application is not located on machines that are potential victims, but it monitors several important links on the network, and tries to detect all attacks. The goal is to identify the IP addresses of all victims and all attackers. To monitor multiple links, the application may be located in multiple capture points.

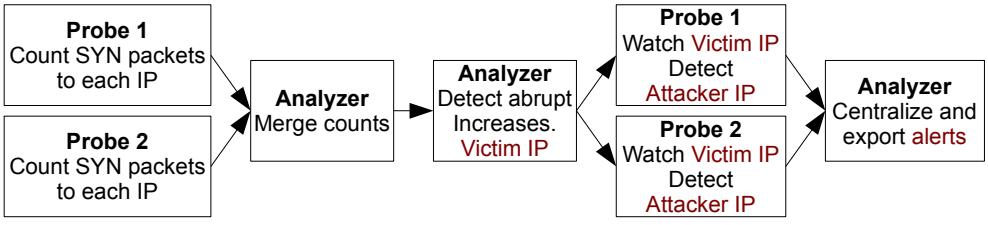


Figure 3.1: TCP SYN flooding detection principle

The method we use to detect TCP SYN flooding is illustrated on Figure 3.1. The process is simplified to describe the major tasks of the detection algorithm. The figure shows only two probes, but the number can vary from one to as many as necessary. The detection process starts by counting the number of TCP SYN

packets sent to each IP address. If an IP address starts receiving a huge number of TCP SYN packets, it is probably victim of an attack. It can then be monitored more closely to check that it is an actual attack and to find the attackers.

Of course, all these tasks are challenging to implement at high data rates. We saw in Section 1.4.1 that it is not easy to maintain one counter per IP address in a way that is memory efficient and provides a fast access. The counters have to be updated at each received TCP SYN packet, and they have to be read regularly for monitoring. The total number of counters can be huge: one counter per IP address seen on the network. In the same way, monitoring a large number of counters to detect abrupt increases is complicated. Monitoring closely some victims to detect attackers is not simple either because few interesting packets have to be extracted from the whole traffic.

So we will now describe the different tools that help implement the process described on Figure 3.1. The first tool is the Count Min Sketch (CMS) algorithm to count packets in a memory-efficient way. The second one is the CUSUM algorithm to detect abrupt changes accurately. The last one is the attackers detection mechanism to make the generated alerts more informative.

Counting packets: the CMS algorithm

The CMS [CM05] algorithm is made to store a large number of counters referenced by keys in a fixed memory. It guarantees short and fixed delays to look-up counter values or increment them. The counterpart is that when reading the value of a counter, an estimate is returned instead of the exact value. But the probability for the estimation error to be over a threshold can be set exactly.

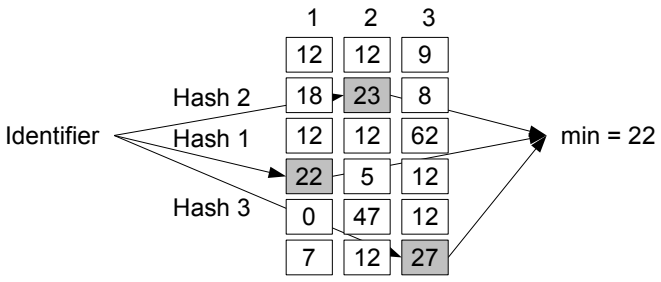


Figure 3.2: CMS algorithm principle with $w = 6$ and $d = 3$

A sketch is represented by a two-dimensional array of integers with width w and depth d. Each integer of the array, noted $count\,[i, j]\,,\ 0 \leq i < d,\ 0 \leq j < w$ is initially set to 0. d pairwise independent hash functions $\phi_i,\ 0 \leq i < d$ are used to hash the keys. These functions take the key as input (32-bit IPv4 addresses for our example application) and return an integer from 0 to $w - 1$.

Update: to increment the counter for key k, for each hash function $\phi_i, 0 \leq i < d$, the integer $count\,[i, \phi_i\,(k)]$ is incremented.

Estimate: to get an estimation of the counter value a_k for key k, the value $\hat{a}_k = \min_{0 \leq i < d}\,(count\,[i, \phi_i\,(k)])$ is computed. Figure 3.2 is an example with

$d = 3$: the minimum of the three values is taken, so the estimated value is 22.

Let $\epsilon = e/w$ and $\delta = e^{-d}$. It is proved [CM05] that $\hat{a}_k \geq a_k$ but also that $Pr\left(\hat{a}_k > a_k + \epsilon \left\|a\right\|_1\right) \leq \delta$ where $\left\|a\right\|_1$ is the sum of all the counters stored in the sketch. So ϵ and δ set the wanted confidence interval and error probability for the sketch. Once these values are chosen, the dimensions of the sketch are set: $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln\left(1/\delta\right) \rceil$. The increment operation requires d hash computations and d increments, the estimation operation requires d hash computations and d comparisons. The memory requirements are the size of $w \times d$ integers.

An important property of the sketches is that they are difficult to revert. It is not simple to know which keys affect a certain value $count\left[i, j\right]$ without computing hashes over all possible keys. It can be an advantage to provide some data encryption, but it can also be a drawback as we will see later.

Another interesting property is that two sketches can easily be merged. Provided that they have the same dimensions and use the same hash functions, merging two sketches $countA$ and $countB$ into a new sketch $countAB$ is as simple as computing $countAB\left[i, j\right] = countA\left[i, j\right] + countB\left[i, j\right],\ 0 \leq i < d,\ 0 \leq j < w$. As all update operations are simple additions, the resulting sketch is exactly the sketch that would have been obtained if all operations made on $countA$ and all operations made on $countB$ had been made on $countAB$. This will help making the DDoS detection application distributed.

For DDoS detection, the CMS algorithm is implemented on each capture point to count the number of TCP SYN packets sent to each IP address through the monitored link. To avoid saturating the sketch and increasing the confidence interval, The full sketch is periodically reset. The reset delay is a parameter that can be fine-tuned depending on the amount of traffic monitored. This way, at each reset, an estimate of the number of TCP SYN packets sent to each IP address over a fixed period is available.

The CUSUM algorithm can then be applied to the sketch to detect abrupt changes. The method used is described in next section.

Detecting changes: the CUSUM algorithm

CUSUM is an algorithm designed to detect change points in a series of values $X_i, i \in \mathbb{N}_{\geq 0}$. For the DDoS detection application, i represents the discrete time and X_i represents a count of TCP SYN packets. CUSUM works with two probability density functions: γ_0 represents the behaviour of the X series before the change, and γ_1 after the change. Detection is made by computing a cumulative sum:

$$S_0 = 0, S_n = \max\left\{0, S_{n-1} + \ln\left(\frac{pr\left(X_n|\gamma_1\right)}{pr\left(X_n|\gamma_0\right)}\right)\right\}, n \in \mathbb{N}_{>0} \tag{3.1}$$

The logarithm is called the log likelihood ratio. It is positive if the probability to get the current value is higher before the change, and negative otherwise. S_n is defined recursively. It is called the test statistic. It increases when the value is more likely under the post-change distribution than under the pre-change distribution and *vice versa*. The test statistic is never less than 0, so that if

the series behaves exactly as it should before the change, S_n remains around 0. Thanks to the log likelihood ratio, S_n increases if the series starts behaving as it should after the change. When the test statistic reaches a threshold h, an alarm is raised to signal that a change happened. The speed at which h is reached depends on the Küllback divergence between γ_0 and γ_1. If the two density functions are similar, the change will be longer to detect. While the threshold has not be reached, the change can always be reverted: if the series starts behaving like before the change again, the test statistic will get back to 0. This mechanism enables a fine-grained control of the change detection, by setting the right threshold.

The drawback of this technique is that γ_0 and γ_1 have to be known so as to detect the change. This is not realistic for DDoS detection because the way the TCP SYN counters will behave during an attack is not known *a priori*. It depends on the way the attack is realised, and attackers often change their methods. The only certain fact is that the traffic will be higher after the change because of the attack. Non parametric CUSUM is a variant that does not use the γ_0 and γ_1 probability density functions. The cumulative sum is computed using only the mean of the values of the watched series before the change (μ_0) and after the change (μ_1), supposing that $\mu_1 > \mu_0$:

$$S'_0 = 0, S'_n = \max\left\{0, S'_{n-1} + X_n - (\mu_0 + \epsilon\mu_1)\right\}, n \in \mathbb{N}_{>0} \tag{3.2}$$

ϵ is a tuning parameter in $[0, 1]$. It moderates the impact of an increase in the value of the time series. Setting it to a high value allows the values of the series to be a bit above the mean, without being considered as an indication that the change happened. ϵ is multiplied by μ_1 to take into account the magnitude of the expected mean after the change. If it is very high, a bigger margin can be left before considering a value as after the change. To guarantee that changes are detected, the mean of $X_n - (\mu_0 + \epsilon\mu_1)$ must be strictly positive after the change, that is to say that $\mu_1 - (\mu_0 + \epsilon\mu_1) > 0$, so we must have $\epsilon < 1 - \mu_0/\mu_1$.

The remaining problem is that the means μ_0 and μ_1 are not known a priori. To fix this, μ_0 is set to the mean of the last N values of the series. This way, the mean before the change is adjusted dynamically. This mechanism avoids raising alerts for slow changes in the network. But μ_1 remains unknown. So $\alpha = \epsilon\mu_1$ is considered as a second tuning parameter. The cumulative sum becomes:

$$S'_0 = 0, S'_n = \max\left\{0, S'_{n-1} + (X_n - \mu_0) - \alpha\right\}, n \in \mathbb{N}_{>0} \tag{3.3}$$

$X_n - \mu_0$ represents the increase compared to the mean. It is moderated by the α parameter, so that increases smaller on the average than α are not taken into account. If the increase is higher, S'_n increases until it reaches the threshold h. If the increase is slow, μ_0 will have enough time to adjust, and S'_n will get back to 0. This way, the CUSUM algorithm is transformed into a self-learning algorithm with only three parameters to set: N is the number of values on which the mean is computed, α is the minimum increase that should not be ignored, and h is the threshold to raise an alarm.

The result of this change detection algorithm is a compromise between the delay to detect a change, and the false alarm rate. With non parametric CUSUM,

under the constraint of a set false alarm rate, the detection delay is proved to be optimal, that is to say as small as it can be [TRBK06, SVG10]. Choosing between a lower detection delay or a lower false alarm rate is just a matter of tweaking the parameters, especially the threshold h.

For DDoS detection, the CUSUM algorithm is used to monitor multiple time series. Values are extracted from the CMS sketch just before the periodic reset. n represents the time: at the n-th period, monitored values are $X_{i,j,n} = count\,[i,j]_n$, $0 \leq i < d$, $0 \leq j < w$. This means that $w \times d$ time series are monitored. The interesting point is that the number of monitored time series does not directly depend on the number of IP addresses on the network. This is good for scalability.

When an alarm is raised on a time series, it is not possible to know directly which IP address is under attack. This is why capture points keep a fixed-size list of the last IP addresses they have seen. When CUSUM raises an alarm on some time series, all IP addresses in the list are tested, until an address is found that impacts cells in the CMS sketch that all raised an alarm. This IP address is considered as under attack.

The raised alarm contains the IP address of a potential victim. In next section, we explain how the attack is then confirmed and the alarm is made more informative.

Finding the attackers

IP addresses are now identified as potential victims of a TCP SYN flooding attack. But the attackers remain unknown, and the alleged attack could even be a simple flash crowd: if a server becomes suddenly popular, a large number of new clients will try to open connections to this server, and the number of TCP SYN packets received by this server will increase.

To confirm the attack and to find the attackers, a closer monitoring of the potential victim is necessary. So for all IP addresses in alert, a new monitoring system is activated. It corresponds to the step "Watch Victim IP" of each probe on Figure 3.1. It filters only packets received by the potential victims, counting for each source IP address, the number of TCP SYN packets and the number of TCP packets with the ACK flag set. These counters are maintained thanks to two CMS sketches. Monitored traffic is unidirectional: from sources to the potential victim. The other way is not monitored because asymmetric routing on the network may prevent the probe from seeing return traffic.

ACK is a TCP flag that acknowledges the reception of a packet. Attackers usually don't bother sending ACK packets, while normal users do. Each time a new SYN packet is received, the rate of SYN packets over ACK packets sent by the IP address is checked. If this rate is over a threshold, and the number of SYN packets is significant, this IP address is considered as an attacker. This confirms that it was a real attack, and not a legitimate flash crowd.

It is then possible to send an accurate alert with the IP address of the victim and the IP address of the attacker. Multiple alerts can be sent for the same victim if there are multiple attackers.

3.3 A flexible framework: BlockMon

Once the monitoring algorithm is specified, it has to be integrated into the the DEMONS architecture. Two goals must be kept in mind: the flexibility of the configuration and deployment of the monitoring application, and the performance in terms of supported data rate. We will now present the development framework provided by DEMONS, in which we will integrate our own application later.

3.3.1 Principles

BlockMon is an open-source [DEM13] part of the DEMONS architecture. In the list of challenges of Section 3.2.1, BlockMon addresses the capture flexibility and speed and the efficient intra-domain communication. It is a framework for the development of applications on network probes. It also handles the communication between these probes and interfaces with a GUI to configure the probes. BlockMon goals are similar to the ones of other monitoring platforms like CoMo and ProgMe [Ian06, YCM11], but it is made to be more flexible, defining no rigid structure of how a monitoring application works. It is also strongly focused on supporting higher data rates than other platforms, despite its flexibility.

The core of BlockMon is developed in C++11 for performance. The version is important because it provides several new mechanisms to handle pointers that improve performance. A daemon developed in Python provides a control interface in command-line or through the network using Remote Procedure Call (RPC). The daemon can start or stop processing on the probe. It can also access to some counter values, or even reconfigure the probe. Configuration is managed using one eXtensible Markup Language (XML) file for each probe.

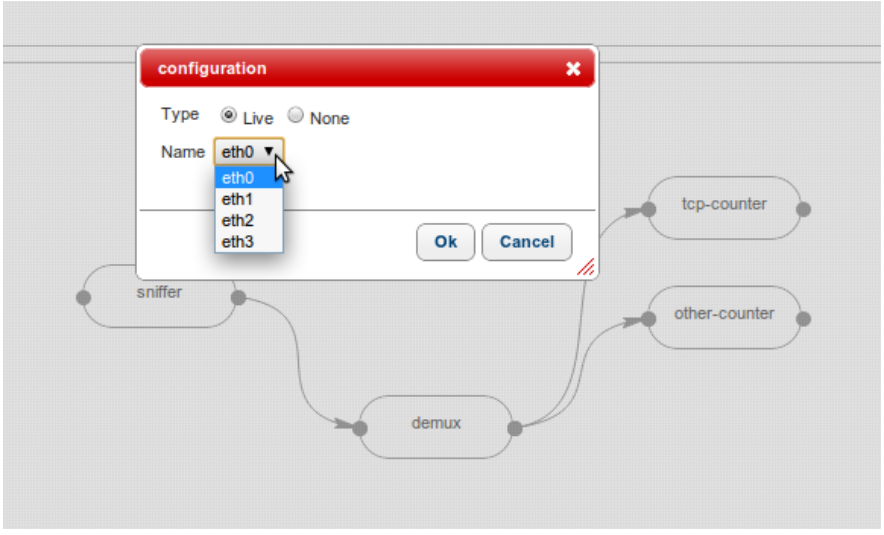


Figure 3.3: DEMONS GUI example: block configuration

The GUI is able to generate the XML configuration file automatically, and to do everything the daemon can do. Figure 3.3 shows an example of how the GUI works. The composition is visible in the background. Blocks can be moved around

and linked using the mouse. The window in the foreground is the configuration of a block. All parameters can be changed the same way as in the XML configuration file. Once the configuration is right, the GUI is able to start and stop BlockMon. It is also able to monitor its status.

A very important aspect of BlockMon is the ability to support high bit rates. As it is very flexible, no network stack is enforced by BlockMon. It can use the default Linux network stack with the *libpcap* library, or it can use the optimized PFQ [BDPGP12b] network stack. Of course, the maximum supported speed is higher using PFQ. The development has been done with performance in mind, so once in BlockMon, packet data is never copied in memory. It is kept in the same location throughout processing and only pointers are moved.

The architecture of BlockMon is based on independent parts of code called blocks. Blocks communicate by sending and receiving messages. Block and Message are two C++ classes that developers can extend. Each block is responsible of a part of the processing. A block can have input and output gates, it listens to messages received on input gates, and sends messages through output gates. A message can represent a packet, a flow of packets, an alert, or any other type of information. When a message is received, the method *receive_msg* of the block is called. This architecture is inspired by the modular router Click [KMC+00], except that messages are not only packets.

The configuration consists in the description of the composition of blocks. Each source file of a block contains a normalized comment section with the list of its input and output gates, the type of messages that can be sent or received through the gates, and a list of named configuration parameters that can be integers, IP addresses, files paths... The XML configuration file is made of a list of block instances with a specific configuration for each instance. A block can be instantiated multiple times with a different configuration. The XML file also contains a connections list. Each connection links an output gate of a block to a compatible input gate of another block. This is enough to configure the behaviour of the probe.

3.3.2 Performance mechanisms

To use the full power of the computer it runs on, BlockMon is multi-threaded. We saw in Section 2.2.2 that the association of an activity to a specific core can impact the performance. To manage this, BlockMon uses the notion of thread pools. A thread pool is a specified number of threads running on specified cores. Each block is associated to a thread pool in the XML configuration file.

Although parallelism is useful for long tasks, all blocks do not need a dedicated thread to run. BlockMon supports three methods of block invocation: asynchronous, direct or indirect. Asynchronous invocation means that the block runs its own loop in a dedicated thread. The *do_async* method of the block is invoked when BlockMon starts and runs continuously. Direct invocation means that the block is only activated when it receives a message or when a timer is fired. It does not have its own thread but runs in the thread of the sending block. Sending a message to a block using direct invocation is equivalent to calling the

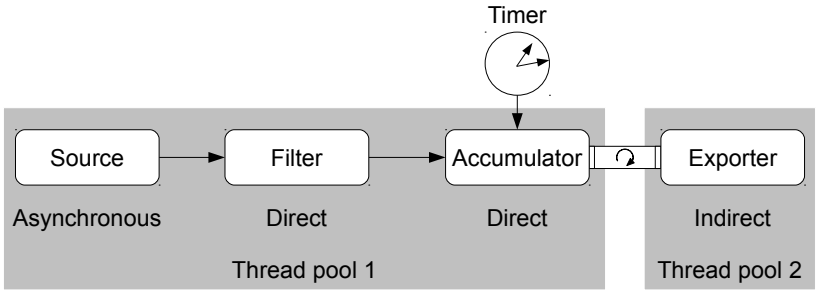


Figure 3.4: Example of BlockMon composition with all invocation methods

receive_msg method of the block. With indirect invocation, the block is still only activated when it receives a message or when a timer is fired, but the *receive_msg* method is called in a different thread from the sending block. Figure 3.4 shows an example of composition with four imaginary blocks using the three invocation methods. The source block is asynchronous because it queries the NIC continuously to get new packets. The filter and accumulator blocks are direct because they perform rapid tasks on each packet: their tasks are executed in the thread of the source block. A timer signals the accumulator to periodically send its data as a message to the exporter. To separate the capture domain that works in real time from the export domain that is not so time sensitive, the export block is indirect.

Messages sent to indirect blocks are handled by wait-free rotating queues. This means that the delay to send a message is only the time needed to write the message pointer in memory. This is important for real-time blocks that cannot waste time sending messages. The scheduler periodically dequeues messages in the queue and invokes the *receive_msg* method of the indirect block.

As explained in Section 2.2.2, memory allocation is a time-consuming task when dealing with traffic at high speed, and the way it is done effects the data access speed. So BlockMon uses two optimizations to make memory allocation more efficient. First, when capturing packets, memory allocations are performed in batches. A batch is able to contain multiple packets. A reference to each batch is kept using a C++11 shared pointer with shared ownership, this way the count of references to each packet is handled by C++ and the memory is freed when all packets in the batch have been destroyed. The second optimization increases the use of cache locality to speed up data access. It works by dividing stored packets into chunks (for example IP header, transport header, start of the payload). All chunks of the same type are allocated space in the same memory location. This way a block that only accesses IP headers will get data from multiple packets in cache.

It is also important to avoid copying data in memory to avoid useless processing. This is why sending a message does not imply copying it. Only a shared pointer to the message is transferred to the receiving block. To do this without having to increment and decrement the reference count, BlockMon uses C++11-specific shared pointers optimizations.

Experiments have been conducted [dPHB+13] that prove that all these opti-

mizations have significant effects on the overall performance.

3.3.3 Base blocks and compositions

Developers can build their own blocks and messages in BlockMon, but many base functions are already provided. The architecture is designed to encourage reuse. The most fundamental blocks are the packet sources. They are asynchronous and read packets continuously by different means. They have one output gate called *source_ out*, through which they send all packet messages. These messages contain the binary data of a packet, and provide commodity methods to parse the most common headers (like Ethernet, IP or TCP). Source blocks include *PCAPSource*, which uses *libpcap* to get packets from a trace file or a live network interface, and *PFQSource*, which gets packets from the optimized PFQ network stack. This block is made to work with high-performance NICs, so it can be configured to read packets from a specific RSS queue of the NIC.

Useful blocks at the other end of the chain are exporters. They are able to convert certain messages into a standardized format and send them through the network. For each exporter, an importer is usually also provided to convert data received from the network back into messages. These blocks can be used by distributed BlockMon nodes to communicate with each other, or they can be used to send data from BlockMon to other applications. Currently blocks exist for the Internet Protocol Flow Information eXport (IPFIX) [CT13] and Intrusion Detection Message Exchange Format (IDMEF) [DCF07] formats. They are called *IPFIXExporter*, *IPFIXSource*, *IDMEFExporter* and *IDMEFSource*. IPFIX is a generic format, it can represent any message. So a developer just has to provide some conversion functions to make a message compatible with IPFIX export and import. IDMEF is a format specific to alerts, so only alert messages are compatible with IDMEF export and import.

Many other useful blocks and messages are available. There is a message representing a flow, another representing an histogram. There is a block to filter packets based on addresses and ports. Another block counts the number of packets and bytes. A block prints packet data to the terminal. Many other available blocks and messages are in the *core* directory in the BlockMon source code.

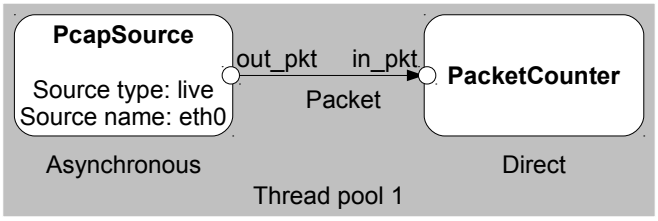


Figure 3.5: A very simple example composition

Figure 3.5 represents a very simple composition that listens on interface *eth0* and counts the number of packets received. It uses *libpcap* to communicate with the NIC. The number of packets can be read using the *PktCount* readable variable.

This can be done directly using the Python daemon, or using the GUI. The GUI can even be configured to generate histograms or other diagrams representing the variable values. Information on the figure is exactly what is described in the XML composition file.

3.4 Implementing DDoS detection in BlockMon

With the architecture of BlockMon in head, it is time to start implementing our own application. The first step is to divide the different tasks of the TCP SYN flooding detection algorithm into blocks. A block is a part of code responsible for a coherent task in a flow monitoring chain, like filtering packets or counting flows. The interest of this division is that blocks can be reused by different applications with a different configuration.

But some parts of our algorithms are so generic, that they can be used for very different purposes and applied on very different kinds of data. These parts are only low-level tools that can be used in different ways by multiple blocks. So they are better implemented as libraries than as blocks. A library provides tools that can be used by any blocks.

For this reason, we started the development of two generic C++ libraries: a CMS library and a CUSUM library. These algorithms are so well-known and generic that they may be used in very different ways.

3.4.1 Algorithm libraries

Count Min Sketch (CMS)

The CMS algorithm is described in Section 3.2.2. For implementation, the chosen hash functions are the same as in the original C code [Mut04]. They are of the form $\phi_i(x) = a_i x + b_i \,[w]$ with a_i and b_i integers. If a_i and b_i never take the same value and a_i is never 0, the functions are pairwise independent hashes. This hash algorithm is very simple and fast, and it fulfills the requirements of the CMS algorithm. It is implemented under the name *ACHash31* in a generic hash library provided by BlockMon.

The CMS library is made to be flexible. It contains two classes. The *SketchId* class represents a key to index the sketch. A maximum length of the key is specified in the constructor. Then bytes can be appended to each other to form the key. Helper functions are provided to append integers and characters. The *CMSSketch* class is initiated with a width, a depth, and a list of hash parameters. Memory is reserved during the initialization. The class provides functions *update* and *estimate* described in Section 3.2.2. Each function takes as argument a key of type *SketchId*. The *update* function adds an integer value taken as argument to the current value of the counter for the given key. The *estimate* function returns an estimate of the current value of the counter for the given key.

A *SketchMsg* message class is also provided in BlockMon core. It enables blocks to send or receive sketches as messages. A bridge class called *IPFIXSketchBridge* also makes this message compatible with the IPFIX export and import

blocks, so that sketches can be sent over the network.

CUSUM

The principle of the CUSUM algorithm is described in Section 3.2.2. It is made to detect changes in time series. For DDoS detection, we use the non-parametric variant of CUSUM described in Equation 3.3 because we assume no *a priori* knowledge about the traffic without and with attacks. To make the library generic, an abstract *Cusum* class is the base of the library. Its single argument is a *threshold*, referred to as h, that is used to decide if an alarm must be raised. The class provides the *check* function that takes as argument the current value X_n of the watched metric. This function should be called periodically. It returns a boolean indicating if an alarm is raised or not.

But the *Cusum* class cannot be used alone, because its *compute_score* method is virtual. It has to be implemented by a subclass. The role of the subclass is to compute the log likelihood ratio necessary to compute S_n from S_{n-1} in Equation 3.1. It takes the *value* X_n as argument. *CusumNP* is a subclass of *Cusum* specialized for non-parametric CUSUM. Its constructor takes as arguments the *mean_window* N and the *offset* α, as well as the *threshold* h. The *check* function value is computed as in equation 3.3 as $X_n - \mu_0 - \alpha$, where $\mu_0 = \sum_{i=0}^{N-1} X_{n-i}/N$.

So after instantiating a *CusumNP* class with the right parameters, a simple periodic call to *check* on the value to watch is enough to run the non-parametric CUSUM algorithm. But for DDoS detection, there is not just one value to monitor. All values of the CMS sketch have to be watched separately, so one instance of the *CusumNP* class is required for each cell of the sketch. The principle used is called multichart CUSUM [Tar05]. To simplify this process, a specific class called *MultiCusum* is made to apply any variant of CUSUM to a list of values instead of just one value. It is initiated with a *Cusum* instance and a *count* of the number of values to watch. The *Cusum* instance is cloned for each value, which means that the computed cumulative sum is fully independent for each variable. The *check* function is applied to a list of values instead of just one value. If at least one value raises an alert, a global alert is returned with the indexes of all values in alert.

3.4.2 Single-node detector implementation

Once basic libraries have been developed, blocks can be implemented for each important function of the TCP SYN flooding detection. To decide which blocks should be implemented, the simplest use case is studied: a detector with only one machine that is used to receive the traffic and to raise alerts. This is an even simpler use case than the one described in Figure 3.1 because there is only one probe instead of two, and the analyzer is on the same machine as the probe.

Figure 3.6 presents the different blocks the detector is made of, as well as the messages used for inter-block communication. All blocks will be described in detail later in this section.

Here is a summary of the global architecture. The first block is *PcapSource*, it simply uses the standard network stack to receive packets and send them as

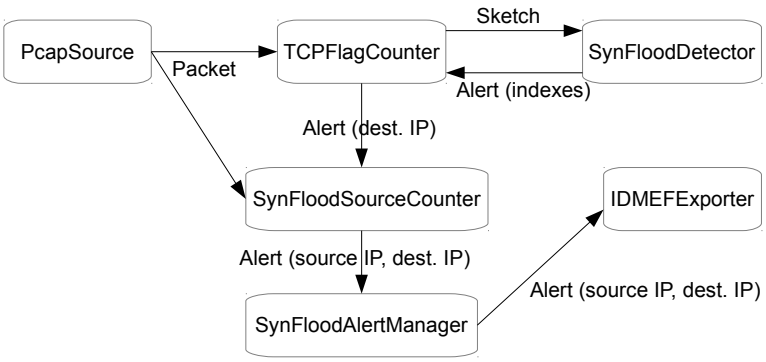


Figure 3.6: Single-node SYN flooding detector composition

messages. The *TCPFlagCounter* counts the TCP SYN packets using the CMS library and exports a sketch periodically. The *SynFloodDetector* uses the CUSUM library to raise alerts with the indices of cells in alert. The alert message is then sent to the *TCPFlagCounter* again to revert the sketch and find the target IP address in alert. The *SynFloodSourceCounter* receives alert messages with the IP address of potential victims and starts monitoring them more closely. It counts the number of TCP SYN and ACK packets sent by each source to the potential victim. If a source sends much more SYN than ACK packets, it is an attacker. An alert is then sent by the *SynFloodSourceCounter* block to the *IDMEFExporter*, which can send the alert using the standard IDMEF format to a configured address.

Below are the detailed descriptions of all blocks of the composition. The *PcapSource* block is not described because it is a basic block provided by BlockMon. The implemented algorithm is exactly the one already described in Section 3.2.2.

TCP SYN counter (*TCPFlagCounter*)

The first step to detect TCP SYN flooding is to count the number of TCP SYN packets to each target IP. This is what the *TCPFlagCounter* block does.

The block instantiates a sketch message provided by the CMS library. It has an input gate *in_pkt*, by which it receives all network packets as messages. When a TCP SYN packet is received, its destination IP address is used as key to increment the corresponding counters in the sketch. A periodic timer is set in the block. Each time it is fired, the sketch message is cloned and sent through an output gate *out_sketch*. The sketch is then reset to count next packets. So exported sketches contain the number of TCP SYN packets sent to each IP address during one period.

As explained in Section 3.2.2, once the CUSUM algorithm has detected the cells of the CMS sketch that raise an alarm, the sketch has to be reverted to find the IP address under attack. So the *TCPFlagCounter* block also has the ability to partially revert the sketch algorithm. It keeps a list of the last destination IP addresses it has seen. It can receive an alert message through the *in_alert* input gate, with the indexes of cells in the sketch. It will then test each recent

IP address to check if it corresponds to these indexes. If an IP address does correspond, it will send the received alert message through the *out_ alert* output gate, but with the destination IP address instead of the sketch indexes.

This block can use direct invocation because it only adds a small processing time for each packet. Its configuration includes the parameters of the CMS sketch (width, depth, and hash functions), and the export period.

Change detector (*SynFloodDetector*)

Exported CMS sketches can be watched to check if a sudden increase in the number of TCP SYN packets sent to an IP address is found. This is the role of the *SynFloodDetector* block.

It has an input gate *in_ sketch*, by which it can receive the periodic sketches sent by the *TCPFlagCounter* block. When a new sketch is received, the *MultiCusum* class of the CUSUM library is used to watch independently each cell of the sketch. If alerts are raised for some cells, the block sends an alert message through its *out_ alert* output gate. The alert contains the indexes of cells in alert. It cannot contain directly the corresponding IP address because the *SynFloodDetector* block is unable to revert the CMS sketch. To get the IP address, the alert has to be sent to the *in_ alert* gate of the *TCPFlagCounter* block.

This block can use direct invocation, processing will be done in the thread of the export timer of the *TCPFlagCounter* block. Its configuration is made of the *CUSUM* parameters (threshold, offset and mean window).

Source detector (*SynFloodSourceCounter*)

Once an alert is available with the destination IP address of a potential attack, one step remains: we must find the source IP address of the attackers. It is also important to check that it is not a flash crowd phenomenon due to the sudden popularity of a server. To do that, the *SynFloodSourceCounter* will monitor more closely all packets to the destination IP addresses in alert.

This block makes heavy use of the CMS library. It has two input gates: *in_ pkt* and *in_ alert*. All packets are received through *in_ pkt*. When an alert is received through *in_ alert*, the target IP address it contains starts being watched. To remember that this address must be watched, a CMS sketch called *filter* is used. The counter for the IP address is increased by 1 when the address starts being watched, and is decreased by 1 when the address stops being watched. This way, the address is watched only if the counter value is not 0. An address stops being watched after a certain delay without receiving an alert. For each watched address, two CMS sketches are created using the source IP addresses as key: one for counting TCP SYN packets and one for counting packets with the TCP ACK flag set. For a normal communication, the number of TCP SYN packets will be lower than the number of packets with the ACK flag set, whereas for an attack, much more TCP SYN packets will be sent.

When a packet is received, the block first checks if it is a TCP SYN or ACK packet. Then it uses the *filter* sketch to check if the destination IP address should be watched. If the address is in the filter, then the corresponding SYN or ACK

sketch is updated. The SYN and ACK counts are then estimated. If the number of SYN is over a configured value, and the SYN to ACK ratio is over a threshold, then the source IP address is considered as an attacker. An alert is then sent through the *out_ alert* gate. It contains the addresses of both the attacker and the victim. All SYN and ACK counts are periodically reset to avoid overflowing the sketches.

This block can use direct invocation. Processing received alerts is fast. Processing received packets is fast for most packets, which are simply dropped. For packets that are currently monitored, it can be a bit longer because it requires to find the proper SYN and ACK sketches, update a sketch, and make estimates for both sketches to check the values. Configuration is made of the periods to reset sketches and forget old alarms, the CMS parameters, the minimum number of SYN packets to check for an attack, and the threshold for the SYN to ACK ratio.

Alert manager (*SynFloodAlertManager*)

Alerts from the source detector contain all required data. The only role of the *SynFloodAlertManager* block is to centralize alerts and make sure that they are not duplicated if multiple probes detect the same attack.

This block receives alerts through an *in_ alert* gate. It keeps a log of a configurable size of all received alerts. If it receives an alert that is already in the log, it ignores it. Otherwise, it forwards the alert through the *out_ alert* gate.

This very simple block should use direct invocation. Its only configuration is the size of the alerts log.

Sketch merger (*SketchMerger*)

An interesting feature of CMS sketches is that two sketches using the same width, depth and hash functions can be summed very simply by summing the values of each cell one by one. This means that the measurements performed by different *TCPFlagCounter* blocks distributed on the network can be combined easily. It can also be used to distribute the load using multiple *TCPFlagCounter* blocks on different threads on the same probe. This merge is the role of the *SketchMerger* block.

It is simply made of one input *in_ sketch* gate, and one output *out_ sketch* gate. Each received sketch is added to an internal sketch kept by the block. When a configured number of sketches have been received, the resulting sketch is sent through the *out_ sketch* gate, and the internal sketch is reset.

This block is very simple. It can use direct invocation. Its only configuration is the number of sketches to merge together.

3.4.3 Alternative compositions

Once all basic blocks have been developed for a simple use case, the flexibility of BlockMon makes it possible to support much more complicated use cases without writing new code. The first use case we have seen is the one of a single machine

running the full detector with a very simple architecture. But an interesting use case to consider is a distributed version with multiple probes and a centralized analyzer. Another important use case is related to the supported data rate. The simple architecture receives packets in a single block, that is to say in a single thread. It could be interesting to use more threads to support higher data rates.

Single-node SYN flooding detector

This is the simple scenario studied in Section 3.4.2. The SYN flooding detector works fully on one machine. Using composition described in Figure 3.6, packet processing and attack detection is made on the same machine in the same instance of BlockMon. All blocks use direct invocation except the asynchronous *PcapSource* block, which captures packets. Final alerts are exported in the IDMEF format using the *IDMEFExporter* block. They are sent through TCP to a configured IP address and TCP port. This architecture is simpler than the one described in Figure 3.1 because there is only one probe and the analyzer and the probe are on the same machine, but the workflow is the same.

Multi-node SYN flooding detector

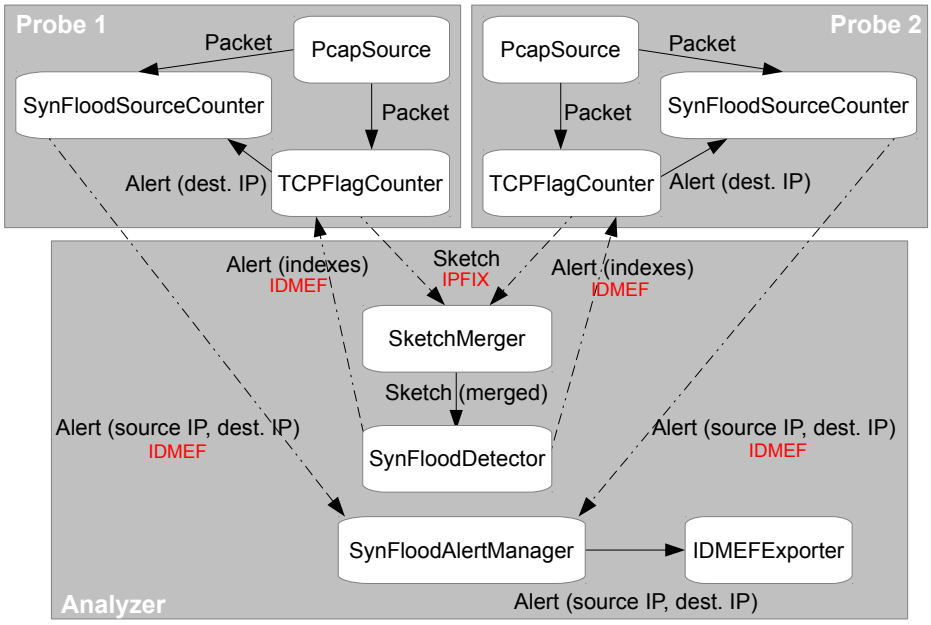


Figure 3.7: Multi-node SYN flooding detector composition

To capture packets in different locations, a more distributed architecture may be necessary. Thanks to the flexibility of BlockMon, changing the XML composition files is enough to get a distributed architecture. Figure 3.7 shows a composition with three machines: two probes and one analyzer. Communication between the machines is made through the network thanks to the exporters provided by BlockMon. Sketches are transmitted in the IPFIX format and alerts

are transmitted in the IDMEF format. Distant connections are represented by dotted lines. The *IPFIXExporter* and *IPFIXSource* or *IDMEFExporter* and *IDMEFSource* blocks used to convert the messages are hidden to simplify the figure. Each dotted line hides one exporter and one source block.

This architecture corresponds exactly to the more high-level Figure 3.1. The communication that was hidden on the high-level figure is the one to revert the CMS sketch once the *SynFloodDetector* block has sent an alert.

Multi-thread single-node SYN flooding detector

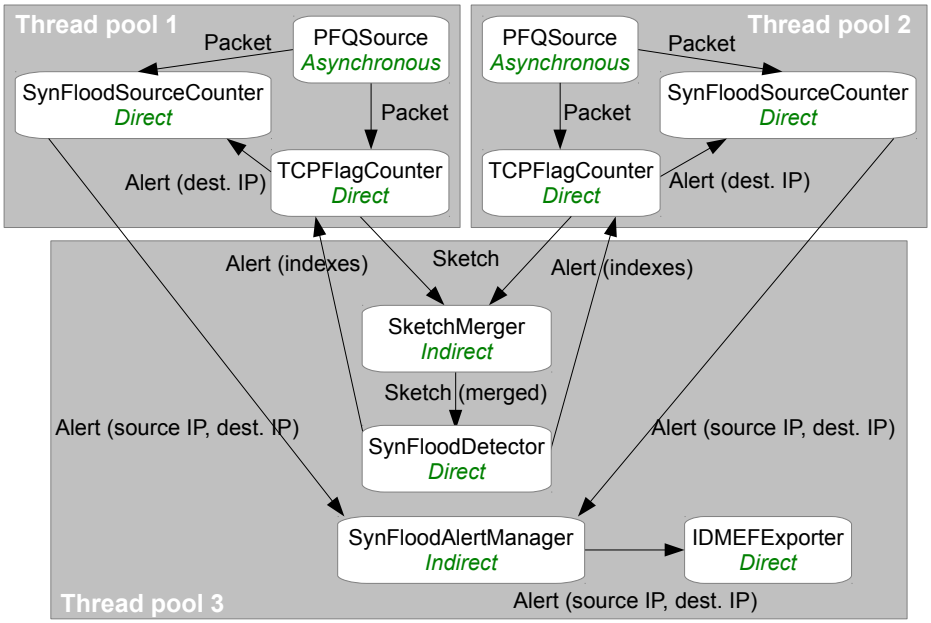


Figure 3.8: Multi-thread single-node SYN flooding detector composition

The same mechanism used to monitor packets on multiple probes can also be used to balance the load on multiple threads on the same machine. Figure 3.8 shows a composition using two CPU cores for packet capture on the same machine. It makes use of the PFQ network stack and the multiple packet queues provided by high-end 10G NICs. Two *PFQSource* blocks are configured to read packets on different queues. For best performance, it is important to make sure that the thread pool of a *PFQSource* block is on the core on which the interruptions for the configured packet queue are received. So it is good to set the interrupt affinity of the network queues manually to control which queue is handled by which core.

All blocks are executed in the same BlockMon instance, so communication is exclusively local. There is no need to use exporters in this composition, except to send the final alerts to mitigation tools. This example uses three cores, two to capture packets and one to analyze the results. But it is very easy to use more than two packet queues, so as to exploit all available cores.

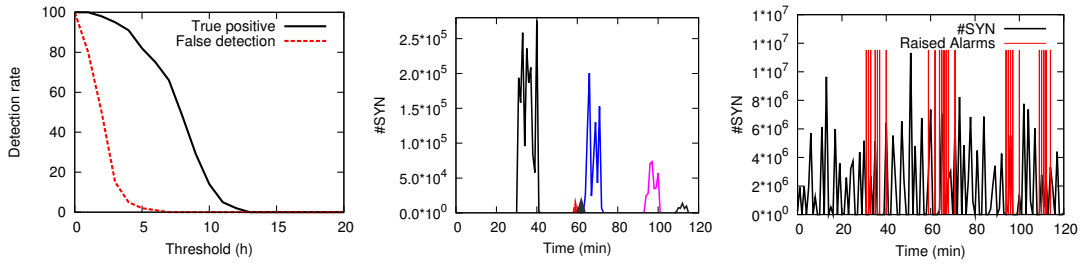
3.5 Results

Thanks to the modular architecture of BlockMon, the DDoS detection application we developed is highly customizable. It can work on a single core on a single machine, it can be distributed over multiple machines, or it can exploit all cores available on a powerful machine to support high data rates. Changing from one architecture to another can be made simply using the configuration GUI.

Tools such as the integration of the PFQ network stack make it simple to get high performance from BlockMon applications, and the optimized blocks and messages architecture provides flexibility without lowering performance. We will now assess the results obtained with our DDoS detection application.

3.5.1 Accuracy

The combination of CMS and CUSUM algorithms we use in our TCP SYN flooding detection algorithm has already been used. Article [SVG10] studies the accuracy of the algorithm using different public traces containing known attacks, as well as ADSL traces obtained in the framework of a French research project called OSCAR.



(a) Threshold variation (b) Attacks in a trace (c) Alarms raised

Figure 3.9: CUSUM accuracy results

Figure 3.9 is taken from [SVG10]. Results are obtained by applying using a software prototype to apply the algorithm to a trace with known attacks. Figure 3.9a presents the percentage of actual attacks detected (true positive) and the percentage of alarms raised that did not correspond to actual attacks (false detection). Varying the threshold h of the CUSUM algorithm is a way to set the wanted true positive and false detection rates. Parameters of the CMS algorithm (width and depth of the sketch) are also important but they are fixed for this experiment. Figure 3.9b presents the number of TCP SYN packets due to known attacks in the trace along time. Figure 3.9c shows the alarms raised by the CUSUM algorithm for the same trace. It can be observed that attacks are all detected and no alarms are raised when there is no attack.

3.5.2 Performance

The important result tested with this implementation of DDoS detection on BlockMon is the performance in terms of supported data rate. We have seen

that BlockMon provides a deep level of flexibility and makes the development of reusable functions easy, but we want to check if it comes at the cost of performance.

The way to separate the load between different cores, the methods to create packet messages that take advantage of cache locality, the message passing without copies and without delays, the support of optimized network stacks like PFQ are all optimizations to make sure that BlockMon will run as fast as possible. But we know that software platforms have difficulties to scale to data rates of 10 Gb/s and more.

To test performance results, we use the composition described in Section 3.4.3 with a varying number of capture threads. The thread pool 3 in Figure 3.8 is not duplicated. It has three available threads on one core. If possible, it should be on a core that is not used for capture to avoid slowing down the capture. Thread pools 1 and 2 are identical. Each pool has one available thread used for capture. Each capture thread is on its own core. In performance tests, the number of capture threads varies from one to eight. The example of Figure 3.8 has two capture threads.

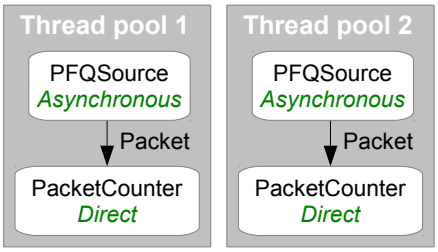


Figure 3.10: Multi-thread single-node packet counter composition

In addition to the DDoS detection application, a much lighter application is tested: a packet counter. This is useful to check if the computation required by the DDoS detection application slows down the processing. The packet counter is also implemented on BlockMon using a default block called *PacketCounter*. The composition for two capture threads is visible on Figure 3.10.

The test machine is an HP Z800 with two Intel Xeon E5620 processors. Each processor contains four cores working at 2.4 GHz. Hyperthreading, a technology that transforms each CPU core into two virtual cores to reduce idle times, is deactivated to make results easier to interpret. The NIC is an Intel Ethernet Server Adapter X520-DA2 with two optical 10 Gb Ethernet ports. To get the best performance from the NIC, the PFQ network stack is used, packets are spread between multiple hardware queues using RSS. Figure 3.11 displays the architecture of the machine, obtained using the *lstopo* tool. Two NUMA nodes are visible, one for each processor. This means that each processor has a privileged access to half of the RAM. It is very visible that two cores from the same processor can cooperate much faster than two cores from different processors, because they have a common cache layer. The two interfaces called *eth2* and *eth3* are provided by the Intel 10 Gb/s NIC. They are connected through PCIe.

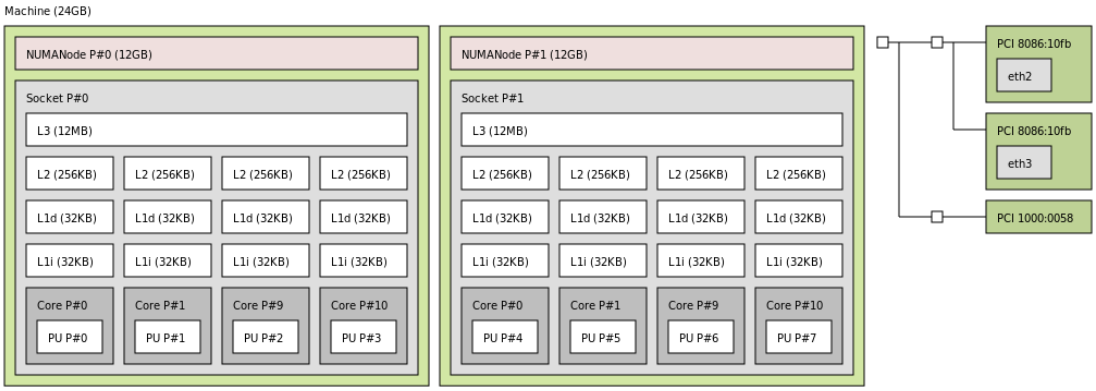


Figure 3.11: Architecture of the test machine

We generate traffic using a commercial traffic generator called Xena [xen12], with two interfaces at 10 Gb/s. Only one interface is used, connected directly to one interface of the Intel NIC. The goal is to stress-test the DDoS detector, so packets generated are TCP SYN packets because their processing is heavier for the monitoring application. The destination IP address is random for each packet, so that different cells of the CMS sketch are updated for each packet. The fact that packets are TCP SYNs has no impact for the simple counter application. For each test, 10 000 000 packets are sent at top speed.

Before obtaining results presented below, a long configuration phase was necessary. Many parameters are accessible on the machine, that have a noticeable effect on performance: the size of the chunks read for the communication with the PCIe bus, the size of buffers used in the network stack and in the NIC, or the CPU frequency adjustment method. Results below use the best parameters we could find. All other NICs on the computer as well as the graphical interface have also been deactivated to avoid slowing down the packet processing.

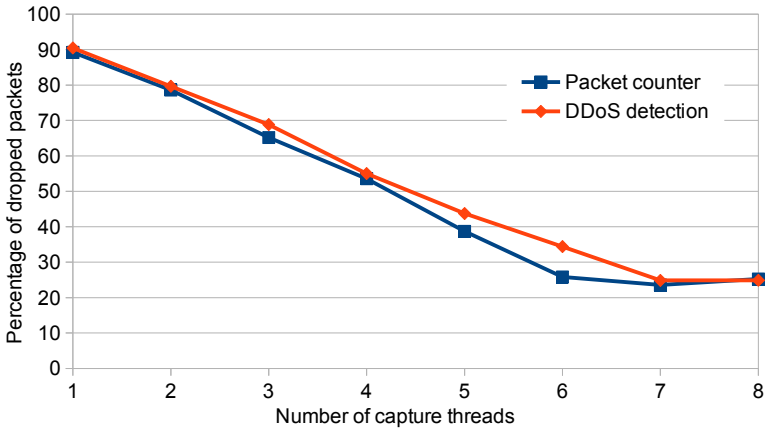


Figure 3.12: Dropped packets with the smallest possible packets

Figure 3.12 displays the percentage of dropped packets using the packet counter

and the DDoS detection application. Packets sent are of the smallest possible size (64 bytes excluding the inter-frame gap and preamble). The number of capture threads varies. The interest of RSS is clear on this figure. Using only one capture thread, the simple counter application drops 89 % of the packets. Increasing the number of capture threads, the number of dropped packets decreases linearly down to 26 % of dropped packets using six threads. With seven threads, results are better with only 24 % of packets dropped, but the improvement is less important. Using eight threads is worse than using seven, with 25 % of packets dropped. It seems that 24 % of packets dropped is the lowest reachable value. Packets are dropped by the NIC and are not sent to the CPU, they appear as *rx_missed_errors* in the statistics of the NIC. This is due to the limits of the communication link between the NIC and the CPU. This limit is not due to BlockMon or to the application. The only way to get better results would be to use a more powerful computer or to find a mean to fine-tune even more the configuration of this computer.

For the DDoS application, results are almost the same as for the counter. The number of dropped packets decreases a bit more slowly, but it reaches the same limit as for the counter. The slower decrease is due to the heavier processing that has to be done for each received packet. The fact that the results are very close shows that the update of the CMS sketch for each packet is not a major problem in software. Results are worse with eight capture threads than with seven. This is due to the same limit as the one observed for the counter. But it is also due to the fact that with seven capture threads, all eight cores are already used because one core is used to centralize all sketches, analyze them and check for alarms. When using eight cores, one core is used both for capture and for analysis, and its resources are shared between all tasks.

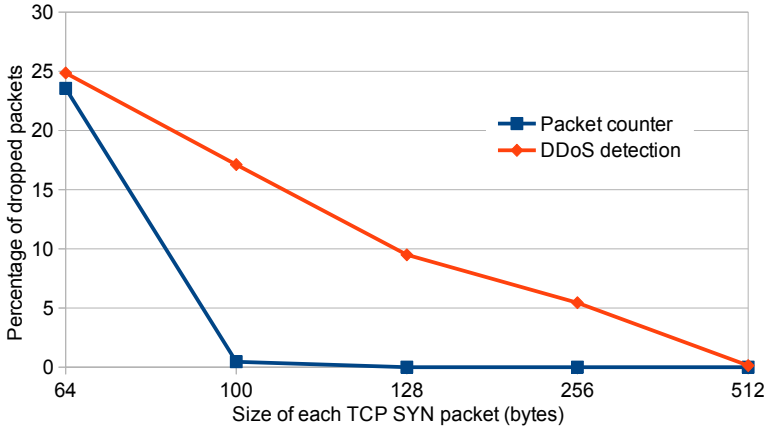


Figure 3.13: Dropped packets using seven capture threads

The fact that some packets are dropped at top speed might be seen as a problem for a real-world use of this software implementation. But the results are for the smallest packets, and the average size of received packets in real traffic is never 64 bytes. Figure 3.13 displays the number of dropped packets depending on the size of the generated packets. Packets are still all TCP SYN, but with

a payload of variable size. As TCP SYN packets carry no data, the payload is only made of zeros. Both the counter and the DDoS detector use seven capture threads as this is the configuration that provides the best results. Results for the counter are simple: with packets of 100 bytes, almost no packets are dropped. And it remains the same with bigger packets. In this case, the delay taken to update the CMS sketch has a more visible impact on performance than with small packets. The CPU is less overwhelmed by packet capture because fewer packets are received per second. The heavier processing of the DDoS detector compared to the counter makes it more difficult to drop no packets but with a size of 512 bytes, the full 10 Gb/s traffic is supported.

These results show that BlockMon is very handy to get the best performance from a computer. The overhead it adds is small enough that with enough capture threads, dropped packets are due to the communication between the NIC and the CPU, and not to the load of the CPU. Supporting 10 Gb/s on a powerful computer is still not an easy task, and packets are dropped under stress test conditions. But a complex application can be implemented in an efficient way in software to support a traffic of 10 Gb/s under normal conditions.

3.5.3 Going further

To improve performance, different approaches are possible. The most basic is to use a more powerful computer, it may allow to support 10 Gb/s even with small packets, but going further will still be difficult. Another approach is to use hardware acceleration.

As the current implementation is pure software, the simplest way to add hardware-acceleration would be to take advantage of the GPU of the computer. But we just explained that performance issues are due to the communication between the NIC and the CPU, not to the processing power of the CPU. So using the GPU would be useless, as the communication between the NIC and the CPU would remain unchanged.

Another option is to use an NPU or an FPGA. Both platforms offer the possibility to support the whole algorithm. As all NPUs offer different features, and as FPGAs are great prototyping platforms, and can be used to design an architecture that will then be implemented on an NPU, we will focus on FPGAs. The CMS algorithm is perfectly adapted to hardware implementation. Each column of the sketch can be a RAM element embedded on the FPGA. This way, all columns can be handled in parallel. If more space is needed, a big external RAM can also be used. The hash functions are easy to compute. We already have implemented the CMS algorithm on FPGA during a project [FGH10] and could reuse this implementation. The CUSUM algorithm is not a problem either on FPGA: it simply consists in some additions and comparisons. It would take some time as all memory cells of sketches have to be read the one after the other, but the delay would be small compared to the period at which the CUSUM algorithm is run. This way, alerts could be raised on the FPGA and sent to the computer, which would export them in the IDMEF format.

Such an implementation would support without problem 20 Gb/s on the

Combo board or 40 Gb/s on the NetFPGA 10G board. But it has a drawback: all the flexibility that was provided by BlockMon is lost. Adding the ability to have distributed probes would require to modify the implementation. Changing the size of sketches would require to reconfigure the FPGA, which would take multiple hours. Another drawback is that development on FPGA is slower that development on CPU, so it should be kept at its minimum. The CUSUM algorithm works without any problem on the CPU, there is absolutely no reason to implement it on FPGA.

The only task that should be offloaded to a hardware accelerator is the lowest-level task: counting packets. This is the only task that has to be performed at wire speed. The change detection algorithm is only run periodically and represents no problem. To keep the flexibility of BlockMon, the best thing to do would be to count packets on the FPGA, and then to do everything else on the CPU using BlockMon. This is possible thanks to Invea-Tech: they developed a BlockMon block that is able to communicate with the Combo board. Two blocks would have to be implemented on hardware: *TCPFlagCounter* and *SynFlood-SourceCounter*. There are two communication channels between the FPGA and the CPU: a very simple configuration channel with a low data rate, and a data channel with a maximum theoric data rate of 20 Gb/s. Sending alerts from the FPGA to the CPU is simple: it represents very few data. The most challenging task is for the FPGA to send periodically CMS sketches to the computer, as the *TCPFlagCounter* has to do. But a sketch may represent just a few hundreds of kilobits. Sending it for example once per second using the data channel is not a problem. It is much less challenging to support for the CPU than receiving the packets. This technique would allow to support 20 Gb/s, even in the worst case. It would require only limited development on FPGA, and it would be almost as flexible as the pure software implementation.

Hybrid solutions using software in collaboration with hardware accelerators are a great way to combine the advantages of hardware and software. The method to do simple low-level processing in hardware and send aggregated metrics to the computer is often efficient. The development cost is higher than for a pure software solution, but performance can be much better.

3.6 Conclusion

In this chapter, we have studied a software implementation of a traffic monitoring application. As part of our contribution to the DEMONS European project, we developed reusable code for the BlockMon framework. This framework is made so that probes can support high data rates. It is developed in C++ and optimized to avoid data copies as much as possible. It integrates with the optimized PFQ network stack, making it compatible with the RSS technology from Intel to separate traffic into multiple hardware queues. It also provides a very accurate control of the way multi-threading works thanks to the notion of thread pools, a set of threads linked to a specific core that can run blocks.

Thanks to the use of configurable blocks, BlockMon is also very flexible and can be configured easily using a GUI. Without any new development, a well-

designed application can be adapted to work on a single probe, or distributed on multiple probes, and to exploit more or less the multi-threading capabilities of the machines. The parameters of the applications can also be tuned.

To test the efficiency of the BlockMon framework, we developed an application for one use case: the detection of TCP SYN flooding attacks. The development was eased by the building blocks that already existed in BlockMon. We implemented two base algorithms as libraries: CMS to count packets and CUSUM to detect abrupt changes. The application itself was divided into multiple blocks to take advantage of the flexibility of BlockMon. All blocks and libraries we developed may be reused for other applications. The libraries are the easiest to reuse because they are very generic. The blocks are designed for a precise use.

To test the ability of our application to support high data rates, we used a traffic generator to send stressing traffic to the DDoS detection algorithm. The application was installed on a powerful computer with a 10 Gb/s Intel NIC. Results show the interest of BlockMon and its multi-threading mechanisms: almost all of the packets were dropped using only one capture thread, but less than a quarter of the packets were dropped with the most stressful traffic possible using seven capture threads. As we saw by increasing the size of the sent packets, traffic on a 10 Gb/s link on a real network should be supported without dropping packets.

The advantage of software development is that it is relatively easy and fast. The flexibility provided by BlockMon would be difficult to offer on dedicated hardware. And the example presented here shows that supporting 10 Gb/s on commodity hardware is possible. But a very powerful machine is required, and its configuration must be fine-tuned to get the best results. With our configuration, we were not able to support 10 Gb/s with the most stressful traffic.

An interesting perspective provided by the DEMONS project is a mix of hardware-accelerated functions and software processing. Invea-Tech has developed a BlockMon block that can communicate with the Combo board. So it is possible to develop some basic processing on FPGA, and then to exploit the flexibility of BlockMon at higher levels. For example for the DDoS detection application, the CMS algorithm is very adapted to an FPGA implementation. So it would be possible to count packets on the Combo board, and then only send sketches periodically to the computer so that BlockMon takes care of the CUSUM algorithm and the data centralization. It would even be possible for some probes running exclusively in software and others using the Combo board to work together.

Chapter 4

Hardware monitoring applied to traffic classification

We have seen the possibilities offered by software-accelerated approaches, as well as their limits using the DDoS use case. This use case is interesting to focus on packet processing, because the algorithm to detect attacks is very simple. It mostly consists in updating counters and comparing them to thresholds. Now we will see the advantages and drawbacks of hardware-accelerated traffic monitoring. But packet processing alone is not a challenge using hardware acceleration, so we will study an application with a heavier algorithmic part: traffic classification.

Traffic classification is the task of associating unknown packets transiting on a link of a network to their generating application. It is used by network operators to get statistics to know more about the traffic that transits on their links. It can also be used to ensure different levels of QoS for different categories of applications, for example by giving a priority to VoIP calls, which require a lower latency than simple downloads. Finally, it is more and more used for lawful interception. In different countries, laws have been in project to force operators to log data about all VoIP calls transiting on their network, which would require operators to identify these calls first. Algorithms we will present can be useful in all these scenarii.

We will first see different algorithms suggested in the litterature for traffic classification, and analyze how they can be implemented to support high data rates in Section 4.1. This will lead us to focus on one algorithm called Support Vector Machine (SVM). Section 4.2 will provide a description of the SVM algorithm, its accuracy, and how it is used for traffic classification. Section 4.3.1 will describe the requirements on the traffic classifier to support high bit rates and show that a software implementation is not enough. The remainder of Section 4.3 will study the SVM classification algorithm and present its challenges and potential. The hardware architecture and implementation of the flow storage and of the classifier will be detailed in Section 4.4, along with a variation of the classification algorithm, which is more adapted to hardware. Finally, Section 4.5 will present both theoretical and real stress-test results using a 10 Gb/s traffic generator.

4.1 State of the art on traffic classification

Different techniques have been used for traffic classification. These techniques have evolved to adapt to the changing nature of the classified traffic [DPC12]. First Internet applications like email, IRC, File Transfer Protocol (FTP) or SSH have been assigned well-known UDP or TCP port numbers by the Internet Assigned Numbers Authority (IANA). Checking only the port number and transport protocol was enough to identify an application. Then P2P applications appeared. The protocols they use are not standardized, and they often use random ports to avoid being filtered. With the evolution of the web, many applications now communicate over HTTP (TCP port 80), be it for file transfers, video or audio streaming, or games. To make traffic classification even more complicated, HyperText Transfer Protocol Secure (HTTPS) is used more and more. For example Google enabled it by default on its homepage. This addition of a security layer means that a bigger and bigger part of the traffic is encrypted, making it impossible to analyze the content of the packets.

All these evolutions explain why traffic classification algorithms are still an open problem, with many different solutions described in the litterature. These solutions can be split into four categories:

Port-based classification is the oldest method. It is based on simply checking the transport protocol and the port number.

Deep Packet Inspection (DPI) is the most wide-spread method. It works by searching for regular expressions in the payload of the packet.

Statistical classification is based on some features computed from the header of one or more packets. It usually applies machine learning algorithms on these features to classify packets into applications.

Behavioral classification uses the same principles as statistical classification, but with higher-level features like the topology of the connections established between IP addresses.

Some traffic classification algorithms actually fall into a fifth category: active classifiers that require the applications to label the packets they generate. We will not study this category because active classifiers have very different use-cases: applications are trusted to label their packets themselves. This is a technique that cannot be used on a public network for QoS or lawful interception, but it can be useful for research to study some generated traffic [GSD+09a].

We will now present algorithms in each of the four categories, and study their ability to support high data rates.

4.1.1 Port-based classification

This is the oldest and the simplest method to classify traffic. The most common transport protocols used over IP are TCP and UDP. Both use 16-bit integers called port numbers to enable multiple applications to communicate at the same

time on the same computer without mixing their packets. A TCP or UDP packet has a source port and a destination port. The source port is reserved on the sending computer for this communication, and the destination port is reserved on the receiving computer. On a classical client/server connection, the communication is initiated by the client. But the client has to send the first packet with a destination port. This is why well-known ports have been created: these port numbers are reserved for a specific application, and servers running an application listen to its well-known port constantly. Standardized protocols have even reserved ports, assigned by the IANA. These ports should not be used by other applications.

Port-based classification consists in finding these well-known ports as destination port for a communication from the client to the server, or as source port from the server to the client. Ports are in a standardized location in the header of the TCP or UDP packets, making it very simple for a classifier to read them. If no well-known port is found, the traffic is considered as unknown.

The CoralReef network administration tool [MKK+01] used this technique in 2001. It did bring good results at the time because most applications used this client/server model with well-known ports. The biggest advantage of this technique is that traffic classification cannot be made simpler, and it can scale to very high data rates. In software, reading a port number is even simpler than transmitting full packet data, so tens of Gb/s could be supported without problem.

Port-based classification was also used in 2003 to identify traffic on a network backbone link [FML+03]. These results are interesting because P2P applications were gaining importance at that time, representing up to 80% of the traffic in some traces, and they are listed in a category "unknown/P2P". The reason is that P2P applications do not use the classical client/server model and use random ports.

In addition to this limit, many applications now use a simple custom protocol over HTTP instead of a protocol directly over UDP or TCP. It has two advantages:

- HTTP is the default protocol for the web so it is authorized everywhere, risks are small to find a private network where this protocol is filtered.
- HTTP directly provides the notion of resources and the handling of server errors. It makes application development faster and more flexible.

This method adds some overhead to the amount of data transfered (custom headers could be simplified), but developers do not consider this as an important problem since connections have become faster. As all HTTP traffic is on the same well-known port, port-based identification of an application that works over HTTP is not possible. Actually, so many applications now work over HTTP that specific classifiers have been designed for this traffic [BRP12].

This is why port-based traffic classification is now considered inefficient [MP05] and other methods have been studied. Although we will see that ports still play a role in some of these methods.

4.1.2 Deep Packet Inspection (DPI)

DPI is currently considered as the most accurate way to classify traffic. It is often used in research as a ground truth [CDGS07] to which new algorithms are compared. It works in two phases:

- First, an expert studies the traffic generated by each application to identify, and finds specific patterns that are found in all TCP or UDP flows generated by an application. These patterns are usually described using regular expressions. This is done offline. At the end of this phase, each application is associated to a signature made of a set of regular expressions.

- Then, online traffic classification is done by comparing the payload of received flows to each regular expression until a match is found. The matching signature corresponds to the identified application.

The basic principle is simple and reliable, so current litterature about DPI focuses on making it simpler to use, for example with a technique to generate signatures automatically without the need of an expert [GHY+13].

But the main research subject about DPI is the acceleration of the online classification. Matching regular expressions against the payload of each received packet is a very challenging task. This is why high data rates can only be reached with specific hardware. For example, an implementation on GPU [FSdLFGLJ+13] manages to reach 12 Gb/s of classified traffic. Different improvements have also been proposed at an algorithmic level [SEJK08, HSL09, CCR11] to build faster pattern-matching automata or to stop searching before the end of the flow for example. FPGA implementations are also available, like this one [MSD+07] with a specific architecture to accelerate pattern matching.

Although DPI is very accurate, it still suffers from the current evolutions of the traffic. Security concerns force more and more developers to encrypt the traffic their application generates. This is for example the case of Google for its search engine and email platform, and Facebook for its social network. In this situation, DPI cannot access to the payload of the packets. The only available piece of information is the certificate used to identify the provider, which will not be enough as more and more providers use encryption for all their services. So the accuracy of DPI is not guaranteed any more [XWZ13]. DPI is also badly perceived in public opinion [MA12] because it means operators study the full traffic their users generate, a fact that raises privacy concerns.

This is why other techniques are studied that do not access the payload of the packets. This is a way to preserve users' privacy, and to make algorithms lighter and more scalable. It also circumvents the problem of traffic encryption, as IP, UDP and TCP headers are not encrypted.

4.1.3 Statistical classification

Many different algorithms [NA08] are used for statistical classification, but they share common points:

- The classification is made by flow. A flow is a set of packets with the same source and destination IP adresses and ports, and the same transport protocol (UDP or TCP).

- The features used to describe a flow are deduced only from the headers of its packets, usually the IP, UDP and TCP headers. These features can be for example packet sizes, inter-arrival times or the number of packets in a flow.

- The algorithm used to classify the flows is based on machine learning. A model is created offline during a learning phase, and the actual classification is made online thanks to the model.

The interest of using only the headers of packets is that statistical classification can work on encrypted traffic [WCM09]. The variety of algorithms comes from the design choices that have to be made.

The first design choice is the set of features used to describe a flow. Some features discriminate different applications better than others. The choice can be made to keep all features and to let the algorithm decide how to separate the applications. Another choice is to select as little features as possible. Automatic algorithms exist [WZA06] for this task. This can make the classification lighter.

The second design choice is the classification algorithm. The machine learning domain is vast and different classification algorithms exist, which are more or less adapted to traffic classification. Each algorithm has its advantages and drawbacks. For example some algorithms are supervised [XIK+12], which means that the classification requires a model built from a data set with flows labeled with the generating application. These data sets are usually generated manually by installing applications and using them, or from a real trace analyzed using DPI or manually. Other algorithms are unsupervised [ZXZW13], which means that they build clusters of similar flows, and use other techniques, often based on DPI, to label each cluster with its generating application. The advantage is that there is no need for a labeled data set, which is challenging to get, but the drawback is that it relies on DPI again, which we try to avoid.

To get the best possible scalability, a proposal uses the Naïve Bayes classification algorithm [SdRRG+12] and the highly optimized capture engine PacketShader [HJPM10]. This way, up to 2.8 millions flows can be classified per second on commodity hardware. But it is proved [WZA06] that the accuracy of this algorithm is lower than SVM or C4.5. In the same way, [JG10] implements the k-Nearest Neighbor classification algorithm on FPGA to classify multimedia traffic, and manages to support up to 80 Gb/s of traffic. Although it is limited to multimedia traffic, it shows the interest of FPGAs, that help scale to higher data rates.

We will focus on two classification algorithms because they proved to have a high accuracy for traffic classification:

- The first one is SVM [CV95]. It has been used successfully in different articles [KCF+08, GP09, EGS09, EG11]. It is a supervised algorithm that uses flow features as vector components and looks for separating hyperplanes

between the flows of each application. A specific function called kernel is used to change the considered space if no hyperplanes can be found. This algorithm brings very good accuracy for traffic classification. Its drawback is that the computation required to classify one flow can be quite heavy.

- The second one is C4.5. It seems to bring a slightly better accuracy than SVM in some situations [LKJ+10]. It is a supervised algorithm based on binary trees. Its main advantage is that it is very light. Learning consists in building a decision tree, with each branch corresponding to a condition on a flow feature. To classify a flow, a condition is evaluated on a flow feature at each level of the tree, and the leaf indicates the application. We will show anyway in Section 4.2.3 that depending on the flow features used, C4.5 can be less accurate than SVM.

SVM brings an excellent accuracy [EG11] at the cost of an increased classification complexity. Classifying a flow using SVM requires the computer to make complex calculations, iterating many times over the kernel function. In [EG11], authors optimize their software implementation of the SVM algorithm to support only 1Gb/s links. A drawback of commodity hardware is that the simple task of processing packets requires a big part of the processing power of the CPU. In [GNES12], authors avoid this problem by using a NetFPGA 1G board to offload packet processing tasks. This way, only the classification is made in software. Using six threads and a simple SVM model with only two applications, they classify 73.2 % of the packets of a trace replayed with a rate of 384 000 flows per second. Section 4.3.1 will give more results about SVM classification speed on commodity hardware, and prove that it is not enough to support a bit rate of only 10 Gb/s.

As SVM is very challenging to implement, it is worth using hardware acceleration to make it scale to higher data rates. Multiple hardware-accelerated implementations of SVM already exist. Although most implementations focus on accelerating the training phase [ABR03], which is not the most important for online traffic classification, some accelerate the classification phase. Some use GPUs [CSK08, Car09], others use FPGAs [IDNG08, PB10, AIN13]. Although GPUs are interesting, FPGAs provide the maximum parallelization possibilities, allowing to explore the best possible hardware acceleration for SVM. Existing implementations [PB10, AIN13] focus on supporting as many features as possible, with a classification problem between only two classes. This is opposed to the statistical traffic classification use-case, where a choice must be done between multiple classes (one per category of applications), and few features are usually available (some packet sizes or inter-arrival times). These approaches use high-speed multipliers of DSP blocks massively to compute kernel values. This is very efficient because DSP blocks are specialized FPGA blocks designed to perform complex computations. But available FPGA boards designed for traffic monitoring like the NetFPGA 10G are not designed to make complex computations, so the integrated FPGA contains few DSP blocks, making these approaches inefficient.

No customized FPGA implementation of SVM has been studied yet for traffic classification. Ideally, such implementation should avoid using DSPs and be adapted to multi-class problems. This is the object of Section 4.4.1. Some articles provide promising ideas about the way to make such implementation. For example in [IDNG08], an algorithm is proposed to compute kernel values using look-up tables, that is to say memory, instead of DSP blocks. In [APRS06], another solution is suggested that modifies the kernel function to make it easier to compute on an FPGA without DSP blocks.

Unlike SVM, the C4.5 algorithm is very simple. Depending on the use case, it can be more or less accurate than SVM. Classifying a flow is only a matter of browsing a tree, with each branch corresponding to a condition. So a software implementation [dRR13] of C4.5 is able to support a traffic of 10 Gb/s, which corresponds to up to 2.8 millions of flows per second. These good results using commodity hardware show the potential of this algorithm. An hardware-accelerated implementation of C4.5 on FPGA for traffic classification manages to support up to 550 Gb/s of traffic [TSMP13].

As a conclusion, SVM and C4.5 have very good accuracy results. We will show in Section 4.2.3 a use case where SVM is better. Although both algorithms are lightweight in terms of data required from the traffic (only some parts of the IP, UDP and TCP headers), SVM is more challenging to scale to high data rates, which makes its hardware implementation particularly interesting.

4.1.4 Behavioral classification

Behavioral classification is based on the observation of the behavior of each host on the network to determine the role of the host, and the reason why a flow is established between two hosts. For example, a host towards which a lot of connections are established is probably a server. By observing the data rate of the flows or their length, it is possible to determine that it is not a web server but a video streaming server. Then all flows communicating with this host on a certain range of ports can be considered as video streaming.

A good example of behavioral classification is BLINC [KPF05]. First a detailed analysis of the behavior of hosts is manually performed for each category of application (mail, P2P, games...), so as to find discriminating connection patterns. Then these patterns are automatically used to classify traffic. For real-time traffic classification, the main challenge is that an important number of connections must first be found with a host so that it can be identified. So first flows cannot be identified. This technique is better suited for offline traffic classification, so that the whole trace can be analyzed before classifying the flows.

Behavioral classification has also been used to classify traffic from P2P video applications [BMM+11]. In this approach, the number of packets and bytes exchanged between hosts is used as feature for an SVM classification algorithm. It gives good results but as for the previous method, a communication must be over before it is classified, so this is better suited for offline classification.

For online classification, an article [BTS06] uses only the size of the first packets of a flow and their direction to or from the server. A flow is seen here

as bi-directional. That is to say that packets with inverted source and destination addresses and ports are considered in the same flow, contrary to previous approaches. This way they reach an accuracy of 90%, which is lower than statistical approaches.

This last method outlines another drawback of behavioral classification: it requires a more global view of the network, which means that data about flows must be bidirectional. This is opposed to the way most statistical approaches work, and it means that classification cannot be done on a backbone, which may not receive both directions of a flow, as routing is not symmetrical.

So behavioral classification is more interesting for offline traffic classification if a global view of the traffic is possible. It has the advantage of being able to identify applications with similar behaviors on the network, and not to require any data about the payload of packets. But it is not the best approach for online traffic classification, both in terms of accuracy and scalability.

4.2 Using SVM for traffic classification

The current state of the art shows that statistical traffic classification is the most promising technique for online traffic classification. Two algorithms give particularly good accuracy results: C4.5 and SVM. SVM is more challenging to scale to high data rates, but it gives better results than C4.5 in some situations (see Section 4.2.3), and a custom hardware-accelerated implementation for traffic classification has not been studied. So we will now present the real implementation of a lightweight classification engine with a high-performance hardware-accelerated solution using SVM.

4.2.1 Proposed solution

Classification is performed on flows. A flow is defined as a set of packets with identical 5-tuples (source IP address and port, destination IP address and port, transport protocol). Flows are unidirectional, so that a TCP session is made of two flows. Each flow is described by simple packet-level features, in this case the size of the Ethernet payload of packets 3, 4 and 5 in the unidirectional flow. The first two packets are ignored because they are often very similar for TCP flows as they correspond to the TCP handshake. This way to use SVM on packet lengths is proved to give good results [GP09, EGS09].

Like any supervised classification method, the SVM algorithm consists of two main phases: a training phase and a detection phase. During the training phase, the algorithm starts from a learning trace labeled with categories of applications and computes the classification model. Using this model, the detection phase decides the application category of new flows.

We consider a hardware-accelerated implementation of the detection phase of the SVM algorithm. The learning phase is done offline in software, as a fast implementation is not required for online traffic classification. Indeed the model might have to be updated once a month, so learning is made offline once a month, but detection is made in real time for each received flow.

Section 4.2.2 gives some background on the SVM algorithm, and Section 4.2.3 shows that the algorithm is interesting for traffic classification in terms of accuracy.

4.2.2 Background on Support Vector Machine (SVM)

SVM [CV95] is a supervised classification algorithm. It transforms a non linear classification problem into a linear one, using a so-called "kernel trick". It takes a set of sample points in a multi-dimensional space, each sample point being associated beforehand with a class. SVM tries to find hyperplanes which separate the points of each class without leaving any point in the wrong class. But it is often impossible to separate sample points from different classes by hyperplanes. The idea of SVM is to use a specific function, called "kernel", to map training points onto a transformed space where it is possible to find separating hyperplanes. The output of the training phase is the SVM model. It is made up of the parameters of the kernel and a set of support vectors x_i that define the separating hyperplane. During the detection phase, SVM simply classifies new points according to the subspace they belong to, using the SVM model. Several algorithms exist for SVM-based classification. We have used the original and simplest C-Support Vector Classification (C-SVC) algorithm [BGV92].

More formally, let us assume that we have a set of training points $x_i \in \mathbb{R}^n, i = 1, \ldots, l$ in two classes and a set of indicator values $y_i \in \{-1, +1\}$ such that $y_i = +1$ if x_i belongs to class 1 and $y_i = -1$ if x_i belongs to class 2. Let us also assume that we have selected a function ϕ such that $\phi(x_i)$ maps training point x_i into a higher dimensional space.

The training phase involves searching for a hyperplane that separates points $\phi(x_i)$ belonging to classes 1 and 2. It solves the following optimization problem:

$$\begin{array}{ll} \min_{w,b,\zeta} & \frac{1}{2} w^T w + C \sum_{i=1}^{l} \zeta_i \\ \text{subject to} & y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i \\ & \zeta_i \geq 0, i = 1, \ldots, l \end{array} \tag{4.1}$$

In the above equation, vector w defines the direction of the separating hyperplane and $\zeta_i, i = 1, \ldots, l$ are slack variables. C is a regularization parameter that penalizes solutions where certain points are misclassified. Once the problem is solved, the optimal w is given by:

$$w = \sum_{i=1}^{l} y_i \alpha_i \phi(x_i) \tag{4.2}$$

α is a vector corresponding to the decomposition of the w vector on the base represented by the $\phi(x_i)$. Only a subset of the α_i coefficients are non-zero, the corresponding $\phi(x_i)$ are the support vectors. They are the only vectors which will be useful for classification.

In the detection phase, any new point x is classified according to the following decision function:

$$\text{sign}(w^T \phi(x) + b) = \text{sign}(\sum_{i=1}^{l} y_i \alpha_i K(x_i, x) + b) \tag{4.3}$$

The kernel function is defined as $K(x_i, x) = \phi(x_i)^T \phi(x)$. x is placed into class 1 if the obtained sign is positive and into class 2 if it is negative.

In this chapter, we first use the Radial Basis Function kernel as a reference, as it was used in article [EGS09]:

$$K(x_i, x_j) = \exp(-\gamma \parallel x_i - x_j \parallel^2) \tag{4.4}$$

Section 4.4.4 presents another kernel, more adapted to hardware implementation, and proves that it brings better performance and accuracy for traffic classification.

From this simple two-class SVM problem, one can easily deal with multi-class SVM classification problems. An usual approach is the so-called "one versus one" (1 vs 1) approach. In this approach $\frac{n(n-1)}{2}$ two-class SVM problems are considered, one for each pair of classes. Training and classification are performed for each two-class problem thus producing $\frac{n(n-1)}{2}$ decisions. The final decision is taken on the basis of a majority vote, that is to say that the new point is allocated to the class which was chosen the highest number of times.

4.2.3 Accuracy of the SVM algorithm

For a given dataset, the accuracy of the SVM-based classifier is measured by the percentage of flows with a known ground truth that are placed in the proper class.

In order to assess this accuracy we have performed validation over three different datasets. The learning and detection phases were performed using the libSVM library [CL11]. The input of the SVM classifier is the one described in Section 4.2.1.

In each dataset, each flow is associated with a label which identifies the application which has generated the flow, called the "ground truth". It has been obtained either by Deep Packet Inspection (DPI), with for example L7-filter [Cle], or by using a tool such as GT [GSD+09b], which is based on the analysis of logs of system calls generated by the different applications and their correlation with traffic dumped on one network interface of a host machine.

Trace	Network of capture	Bytes	Flows	Known flows	Rate (Mb/s)
Ericsson	Local Area Network, Ericsson Laboratory	6 222 962 636	36 718	16 476	3.96
Brescia	Campus trace, University of Brescia	27 117 421 253	146 890	74 779	103
ResEl	Campus trace, Télécom Bretagne	6 042 647 054	25 499	10 326	176

Table 4.1: Traffic traces and their properties

The characteristics of the three traffic traces used as benchmarks are listed in Table 4.1. Known flows are flows for which the generating application is known. Traces correspond to three different scenarios: one laboratory environment, one

campus network, and one student residential network. As a consequence, the composition of traffic is significantly different from one trace to the other.

1. The Ericsson dataset corresponds to some traffic that has been generated in a laboratory environment at Ericsson Research.

2. The Brescia dataset is a public dataset [GSD+09b]. It corresponds to traffic captured on a campus network. The ground truth has been obtained with the GT tool.

3. The ResEl dataset is a trace we captured ourselves on the network for student residences at Télécom Bretagne. ResEl is the association managing the network, which agreed to the capture on the single link between the residences and the Internet. It was performed around 1 PM on a workday for a bit more than 5 minutes with an average data rate of 84 Mb/s. The ground truth was obtained using L7-filter. The GT tool was not used, because it would have required installing probes on the computers of certain students, which was not possible.

As each trace corresponds to a completely different network, the parameters of the SVM algorithm have been set differently, and one different SVM model has been learnt for each trace.

The definition of classes is not universal. It mainly depends on the filters that have been defined for DPI. In order to enable a comparison between traces we have merged applications into different categories, listed in Figure 4.1. An order of ten classes is classic. Unsupervised classifiers produce more classes, but this is an undesired feature.

Using the RBF kernel and the best parameters found in cross-validation, the accuracy, that is to say the overall percentage of flows that are correctly classified is 97.82 % for Ericsson, 98.99 % for Brescia and 90.60 % for ResEl.

A global accuracy figure is usually not considered sufficient to demonstrate the performance of a classifier. Some classes could be frequently misclassified with not much impact on the global figure if few flows correspond to those classes. A usual representation of results is given by the confusion matrix. Figure 4.1 provides the accuracy per application category, that is to say the percentage of flows of each application category that has been accurately classified.

As one can see from this figure, the accuracy of the SVM algorithm differs from one application category to another and from one trace to another. The proportion of an application category in a trace impacts the ability of the SVM algorithm to detect it. For example, as the "Web" class is present with a good proportion in all three traces, the accuracy of the detection is high. However, as the "Streaming" class, is almost absent in the three traces, it has the worst classification accuracy. Some classes are completely absent from certain traces. For example no gaming applications have been detected in the Brescia and ResEl traces, maybe because recent games were not detected when applying DPI with too old signatures.

Accuracy for the ResEl trace is not as good as for the other traces. This might be due to the way we got the ground truth, simply using L7-filter instead of more

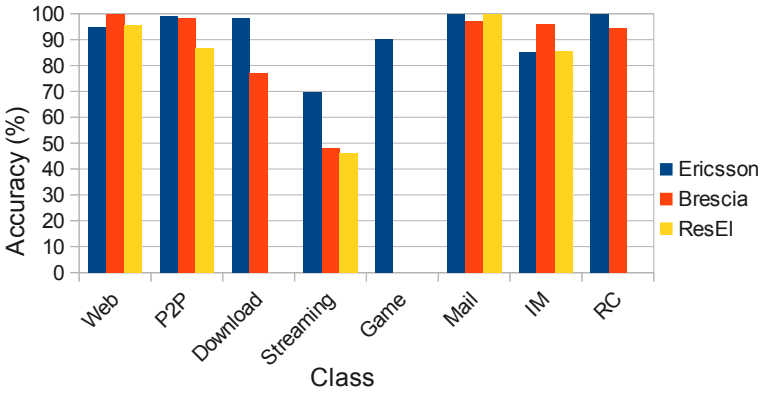


Figure 4.1: Accuracy per traffic class

elaborate tools like GT: some flows may be misclassified in the ground truth. But this represents real-world applications where the ground truth is always difficult to get, so we keep this trace to check that an implementation works on it as well.

As discussed in Section 4.1.3, C4.5 is an algorithm known to give good results for traffic classification, even better than SVM in some situations [LKJ+10]. So we tried to use C4.5 on the Brescia dataset, using two third of the trace for testing, and a part of the remaining trace for learning. C4.5 always resulted in more classification errors than SVM, with 20% more errors using a third of the trace for learning (16 602 flows), and even 45% more errors using a tenth of the trace (498 flows). This may be due to the fact that we do not use ports as classification features.

The same article [LKJ+10] also claims that using discretization improves the results of SVM classification. Discretization consists in grouping the feature values (here the packet sizes) into intervals. Classification is then done using these intervals instead of the values. We tried using the entropy-based minimum description length algorithm to discretize the features of the Brescia dataset. We obtained a model with fewer support vectors, which is interesting in terms of classification time, but with almost three times more classification errors. So the implementation described below uses SVM without discretization, although it would be simple to add.

4.3 SVM classification implementation

4.3.1 Requirements

In the following, the implementation of on-line SVM traffic classification is studied. In our scenario, probes are located on an operator access or core network to monitor traffic at packet level, reconstruct flows and classify them with an SVM. Only the detection phase of SVM is performed on-line. The learning phase is performed off-line periodically with a ground truth generated by tools such as GT. Using boards such as the NetFPGA 10G [Net12] or the COMBO LXT [IT13], algorithms should be able to support up to 40 Gb/s. But the eventual

goal for the algorithm is scaling to 100 Gb/s and more.

Two main functions will be required for traffic classification:

- The flow reconstruction reads each packet, identifies which flow it belongs to, and stores the packet lengths required for classification. The processing speed depends on the number of packets per second in the traffic.

- The SVM classification runs the SVM algorithm once for each received flow. The processing speed depends on the number of flows per second in the traffic.

The traces used to test the classification algorithm are described in Table 4.1. To reach a specified bit rate, requirements in terms of packets per second for flow reconstruction and flows per second for classification are different depending on the dataset considered. This is due to different average packet and flow sizes. The flow size is the number of bytes received between the reception of two flows with enough data to send them to be classified. It includes possible delays between packets. Table 4.2 provides these numbers for each trace. So for example, to support a data rate of only 10 Gb/s, the flow reconstruction speed should range from 1 420 313 to 1 921 405 packets per second and the SVM classification from 6 771 to 9 054 flows per second. The ResEl trace is the easiest to handle for flow reconstruction because it contains large packets, but the hardest to handle for SVM classification because it contains small flows.

Trace	Average packet size (B)	Average flow size (kB)
Ericsson	651	169
Brescia	812	185
ResEl	880	138

Table 4.2: Average packet size and flow size for each trace

We first developed a software version of the classifier that is fed by a trace, to assess the possible performance in software. The classifier is made up of 3 main modules: (i) reading the trace, (ii) rebuilding flows from the stream of packets, and (iii) classifying flows. Flow reconstruction is handled using the C hash table library, Ut Hash [Han13]. For the SVM algorithm, the libSVM [CL11] library (written in C) was chosen. To use all the cores of the processor, openMP [DM98] for libSVM is enabled.

Table 4.3 shows the performance of the software implementation on a 2.66 GHz 6-core Xeon X5650 with hyper-threading enabled and 12 GB of DDR3 RAM. Performance values are derived from the time it takes the flow builder and classifier modules to process the whole trace, divided by the number of packets and flows in the trace. Maximum speeds are computed using the mean values in Table 4.2. It shows that the software implementation is only able to support low speeds, under 10 Gb/s. The best supported speed ranges from 2.7 Gb/s to 8.1 Gb/s depending on the trace.

The flow reconstruction speed does not really depend on the trace, as the flow reconstruction algorithm requires a time that is almost constant per packet due to the hash table.

SVM classification is always more limiting than flow reconstruction (with a maximum speed of 2.7 Gb/s in the worst case). Its speed depends on a variety of factors, including the number of support vectors in each SVM model: Brescia is the trace for which the learnt model has the most support vectors (14 151), followed by the ResEl (5 055) and Ericsson (3 160) traces.

Trace	Packets per second (flow reconstruction)	Flows per second (classification)
Ericsson	5 214 218 max. 27 Gb/s	5 975 max. 8.1 Gb/s
Brescia	5 531 895 max. 36 Gb/s	1 827 max. 2.7 Gb/s
ResEl	4 750 621 max. 33 Gb/s	5 619 max. 6.2 Gb/s

Table 4.3: Performance of the software implementation

Even with a powerful computer, a software implementation is not able to reach 10 Gb/s, mainly due to its limited ability to parallelize the computation. This justifies the use of hardware acceleration. Different platforms may be used to provide hardware acceleration for network monitoring:

- Graphics processing units can help accelerate the algorithm, but not the packet processing.
- Network processors are programmable in software and provide hardware-accelerated tools for tasks commonly required in network monitoring. But platforms are proprietary, development is not portable, and each commercial solution has very different performance.
- Programmable cards with an integrated Field-Programmable Gate Array (FPGA) are very flexible and provide hardware access to network interfaces. Although development time is long, the code is portable and the results do not depend on a specific vendor.

To be able to fully explore the parallelism possibilities in the SVM classification algorithm, we have chosen to use a board with an FPGA that is more flexible than network processors. Two main vendors provide such boards: (i) NetFPGA with the NetFPGA 10G, which has four interfaces at 10 Gb/s and a Xilinx Virtex-5 XC5VLX155T FPGA, and (ii) INVEA TECH with the Combo, which has two to four interfaces at 10 Gb/s and a Xilinx Virtex-5 XC5VTX240 FPGA. The FPGA provided by the NetFPGA 10G performs better than that on the Combov2 board, so we will perform synthesis on the NetFPGA 10G FPGA. But for now the only 10 Gb/s board we have is a Combo, so stress tests will be

made on the COMBO-LXT board with a COMBOI-10G2 extension (two interfaces at 10 Gb/s).

Before studying the actual implementation of both the flow reconstruction and the SVM classification, next section will check that the SVM algorithm can benefit from hardware acceletion.

4.3.2 The SVM classification algorithm

The classification part of the SVM algorithm takes a vector as input and returns the class of that vector as an output. The main part of the computation is repeated for each support vector. Algorithm 1 describes the steps of this computation. It is the multi-class implementation of the decision making procedure described in Section 4.2.2 equation 4.3. This pseudo-code has been written in order to enlight the possibilities for parallelizing the algorithm.

Algorithm 1 SVM classification algorithm

1: $x \leftarrow$ the vector to classify
2: **for all** support vector x_i **do** {Main loop}
3: $c_i \leftarrow$ the class of x_i
4: $k_i \leftarrow K(x_i, x)$
5: **for all** class $c_j \neq c_i$ **do** {Sum loop}
6: $d \leftarrow$ index of the decision between c_i and c_j
7: $S_d \leftarrow S_d + y_{d,i} \times \alpha_{d,i} \times k_i$
8: **end for**
9: **end for**
10: **for all** decision d between c_i and c_j **do** {Comp. loop}
11: **if** $S_d - b_d > 0$ **then**
12: Votes $V_i \leftarrow V_i + 1$
13: **else**
14: Votes $V_j \leftarrow V_j + 1$
15: **end if**
16: **end for**
17: Select class $c_n \leftarrow$ class with the highest votes V_n

The support vectors and the y, α and b values are parts of the SVM model. Compared to the notations used in Section 4.2.2, index d is added to identify the binary decision problem considered for the model values.

4.3.3 Parallelism

The Main loop is where most of the computation time is spent. It iterates many times (from 3 160 to 14 151 support vectors for the traces presented in Table 4.1) and includes complicated operations (exponential, multiplications). But it can be easily parallelized, as each iteration is independent of the others. The only shared data is in the additive S values. These values have to be duplicated so that iterations are computed in parallel. Then, as S is additive, the duplicated values can be merged by adding them together.

The Sum and Comparison loops have few iterations: one per class. For our traffic classification problem, there are eight classes defined in Figure 4.1, so the loops can be totally parallelized. The Sum loop iterations are totally independent, while the Comparison loop iterations share the vote counter, which is additive. So it can be duplicated and then merged.

All loops can be removed by using parallel processing except the main loop, which has too many iterations and would require more area than is available on the Virtex-5. But it is possible to implement the Main loop more than once, so that fewer iterations are required to process one vector. Section 4.4.1 describes an architecture with an adjustable level of duplication of this loop.

4.4 Adaptation to hardware

4.4.1 Architecture

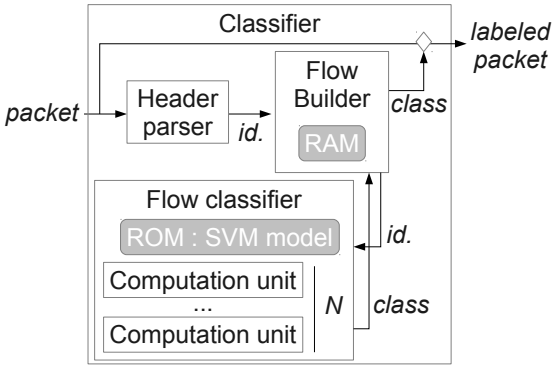


Figure 4.2: Architecture of the classifier

The architecture of a traffic-processing module on NetFPGA or Combov2 boards is very similar. It uses a block with an input bus for input traffic, and an output bus for output traffic. The classifier block is described in Figure 4.2. It is divided into different blocks:

The header parser just reads the packet headers to get the flow identifier (source and destination IP addresses, transport protocol, source and destination ports) and the size of the packet. The current implementation supports only IPv4 over Ethernet, so it is very basic. Supporting IPv6 would simply change the flow identifier, which includes the source and destination IP addresses. It would not change the design fundamentally.

The flow builder contains the list of all current flows. It is a big memory in which a lookup is made for each received packet. It contains for each flow a state (number of received packets, classification running or done), the three packet sizes used for classification if known, a timestamp and the class if already known. If the class of the flow is already known, the packet is labeled and transmitted. Otherwise, the received packet size is stored in

memory, and if enough data is available, a classification request is sent. The packet is transmitted without label.

The flow classifier is made up of multiple computation units. It is the most important part of this architecture. It implements the computation of the main loop described in Algorithm 1. To get the best performance from the FPGA, the operations of the algorithm must be parallelized. As seen in Section 4.3.3, all loops except the main loop can be totally unrolled by duplicating the hardware for each iteration. The computation unit is duplicated, as much as the FPGA supports.

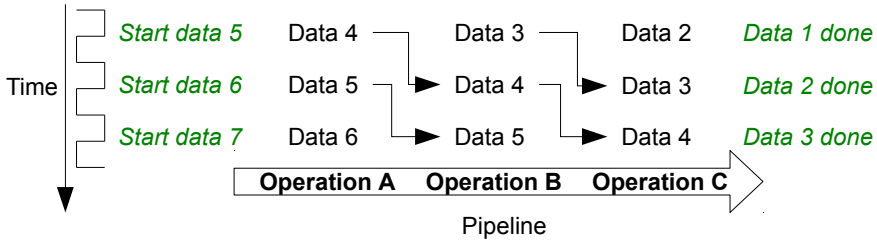


Figure 4.3: Pipeline with three operations A, B and C during three clock cycles

As the computation in the main loop is complicated, each iteration will take many clock cycles in hardware. To improve the throughput of the loop and reduce its computation time, the iterations can be pipelined. The principle of a pipeline is described on Figure 4.3 with three operations A, B, and C. Each operation depends on data computed during the previous operation, so they cannot be fully parallelized. But with a pipeline architecture, one new data item is sent for computation to each unit at each clock cycle. Data item 4 is first sent to operation A, while operation B processes data item 3 and operation C processes data item 2. At the next clock cycle, operation B can compute data item 4, as operation A has completed. This way, one new data item enters the pipeline at each clock cycle, and one new result exits the pipeline at each clock cycle. All operations work in parallel. For the main loop, each data item is a support vector, so computation starts for one new support vector at each clock cycle.

As Figure 4.2 shows, the SVM model is currently stored in ROMs. This makes it necessary to synthesize the design again to change the SVM model used. If more flexibility is required, ROMs can be converted into RAMs, so that it is possible to change the SVM model faster. This should not require the implementation to be changed, but might complicate the placement of the logic on the FPGA, as new connections will have to be created to write into these memories.

In this section, the design of the flow builder will first be described. Then two different implementations of the computation units will be studied, as well as the advantages and drawbacks of each method.

4.4.2 Flow reconstruction

Requirements

Flow reconstruction is made in the flow builder block. It consists mainly in managing an external RAM to store data about each flow and find it back when a new packet is received. Data stored is made of:

- A status field to know if the flow is incomplete (less than five packets), being classified, or already classified. 3 bits are needed for this.
- The three packet sizes needed for classification. Packet size can be up to 1500 bytes, so it is stored on 11 bits. For the three sizes, 33 bits are required.
- A timestamp to know when the flow timed out and may be erased. 8 bits are enough to store it. A resolution of one minute is enough, and more than one hour is needed to get back to the same timestamp using 8 bits. Flows do not stay that long in memory.

So at least 44 bits are required to store data about one flow.

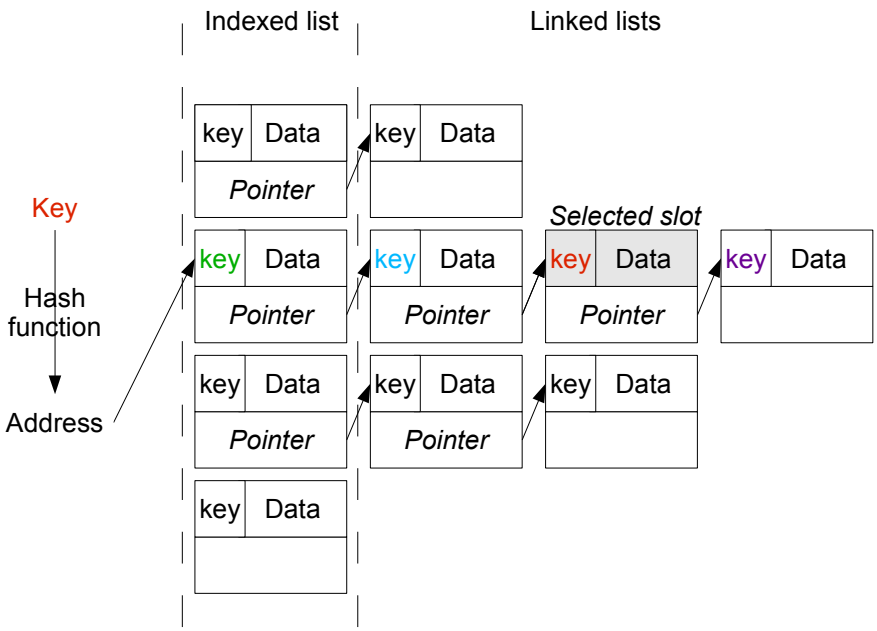


Figure 4.4: Classical hash table look-up principle

As explained in Section 1.4.1, the classical solution to store this kind of data is a hash table [SDTL05]. The principle is illustrated on Figure 4.4 : each stored element has an identifier, called a key. When an insertion request is received, a hash function is used to transform the key into an index in a list of slots stored in memory. If the selected slot is not used, the element is stored in the slot. If it is used, a collision arises. In this case, an algorithm is used to find another slot for the element. The algorithm can for example be a linked list: each slot

stores an element and a pointer to another slot. The last slot of the linked list has an empty pointer. To add an element, a new slot is created by allocating memory, and the pointer of the last slot is updated. During lookups, the pointers are followed until the good key or the end of the list is found.

A lot of research has been done on improving hash tables, reducing their read and write latencies and their memory requirements. But to ensure that a new key can always be saved somewhere, hash tables all have two common drawbacks. The first one is that the space they use is variable because a slot may have to be added to any linked list, requiring new memory to be allocated. The second drawback is that although the insertion and lookup delays are most of the time very low, no sensible maximum can be guaranteed because of possible cascaded collisions. If all elements end up in the same linked list, a number of read operations equal to the number of stored elements may be required for one lookup.

For flow reconstruction on FPGA for online traffic classification, some requirements are mandatory:

- To optimize performance, we use all the external RAM available on the board for flow reconstruction. So the memory space has a fixed size and cannot be increased when needed. For example there are two 72 Mb QDR-II SRAM memories on the Combo board. So two times 1 048 576 slots of 72 bits can be addressed. Both memories can be used in parallel.

- To tag packets in realtime with their application category, at each received packet, a flow lookup is sent, and it must return before the packet can be released. So the flow lookup time must be guaranteed and constant. A slow lookup would mean that the packet is delayed on the link, and a variable lookup time would increase jitter. Delay and jitter are essential QoS features on a network, so an equipment that will be used on a real network must give guarantees about the degradation of delay and jitter.

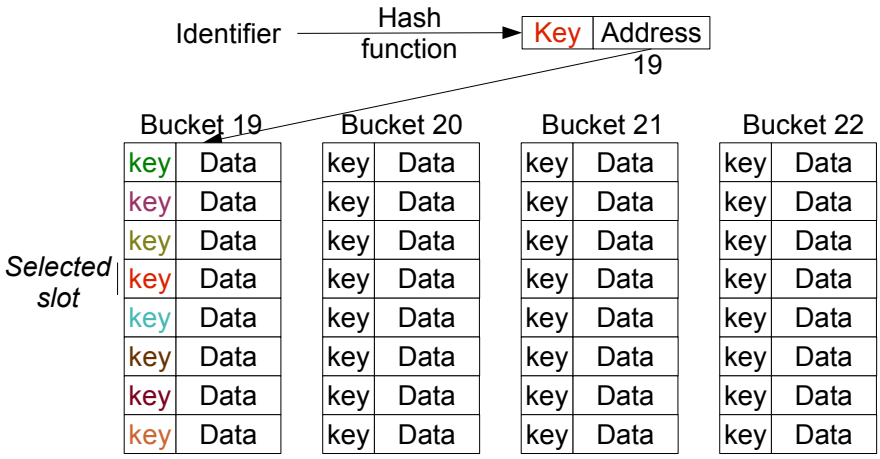


Figure 4.5: Bucket-based algorithm principle

To attain these goals, an implementation of NetFlow on NetFPGA [Mar08] uses a simplified hash table described on Figure 4.5 : the available memory is

divided into buckets of eight slots. Each slot is big enough to store data about one flow. A hash is computed from the flow identifier. It can work with many hash functions. An easy-to-implement hash function is discussed in Section 4.4.2. The first part of the hash is used as an address to select a bucket. The second part of the hash is used as a key, smaller than the original identifier, so as to waste less space in memory than the full identifier. The key is stored in the selected slot with flow data. This way, during lookup, the key is compared to the value stored in each occupied slot of the corresponding bucket. If the same value is found, flow data is returned, otherwise the flow is unknown. If no space is found during insertion, a random slot is chosen and the previously stored flow is lost. Using this method, insertion requires up to eight read and one write operations, and lookup up to eight read operations. The unavoidable cost of a fixed memory size and lookup delay is a non-zero loss probability.

Although this method is very simple to implement and fast, if more than eight concurrent flows are in the same bucket, some flows will be lost. We performed a simulation detailed in next section with 2 097 152 slots (the number of 72-bit slots on the Combo board). It shows that with one million concurrent flows, 0.66% of the flows are lost (Figure 4.7). This is an important loss considering that the number of available slots is double the number of flows to store.

The storage algorithm

To get better results, we used a principle based on the CMS algorithm described in Section 3.2.2. CMS has been designed to store a large associative list of counters using a constrained memory space. It is highly adapted to hardware implementations. The principle is to use multiple uncorrelated hash functions to compute different hash values from an identifier. These hash values are used as addresses in the memory. Each address points to a counter value. When incrementing a counter, each counter pointed by one hash value is incremented. To get the value of the counter, the minimum of all pointed counters is taken. The idea is that collisions are accepted. If at least one of the pointed counters encountered no collision, taking the minimum will give the good result.

For flow reconstruction, stored data is not only made of counters but the same principle can be used. Here is the adapted algorithm, illustated on Figure 4.6:

Eight independent hash functions are used. Each hash value is split into two parts: the first part is used as the address of a slot in memory, and the second part is used as a key stored in the slot with flow data. For the SRAM on the Combo board, 22 bits of the hash are used as address, and a memory slot is 72 bits wide.

During a lookup, each of the 8 slots are read. The first one with a stored key equal to the one computed by the hash is a valid value. It is returned.

During an update/insertion, each of the 8 slots are read. Then multiple slots are selected for writing: the slots with the same key if it is an update, the empty slots and the slots with a timestamp older than a set threshold. This timeout threshold impacts directly the number of concurrent flow. It must

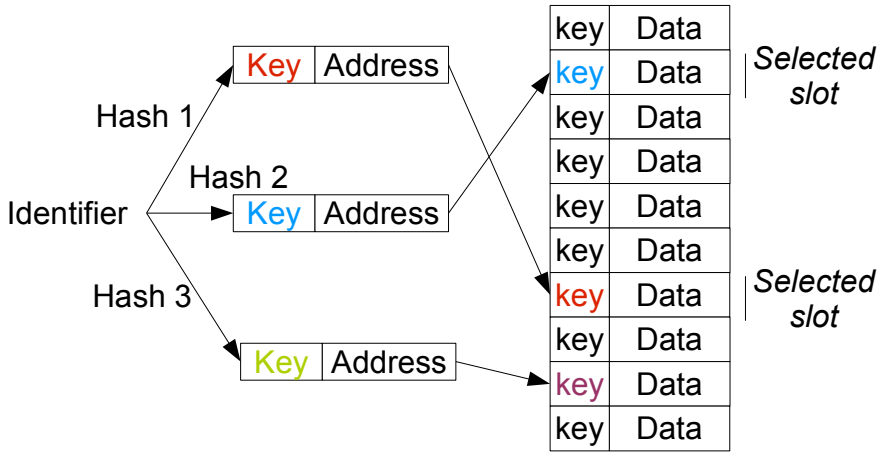


Figure 4.6: CMS-inspired algorithm principle

be set to a big enough value so that most flows with no new packets during this delay are actually over, but small enough so that old flows are not kept too long. Before writing, a new field is added to flow data: the replication level. This is the number of slots that have been selected for writing. This field will be used for future insertions if no slot is available: then one of the slots with the highest replication level will be overriden. This way the chances for this deleted flow to still be stored somewhere else are higher.

This method uses up to eight read operations for a lookup, and eight read operations and up to eight write operations for an update or insertion. The way it is implemented in hardware, it always uses the time of eight read operations for a lookup, and eight read and write operations for an insertion. For an update, an optimization is used: as a lookup always happens before an update, information on where data should be stored is returned by the lookup process, and is used by the update process to avoid the eight read operations.

Another hardware optimization is used: on the Combo board, two SRAM memories can be used in parallel. So four hash functions are used to address the first memory, and the four other hash functions are used to address the second memory. This reduces a bit the diversity of the hash values, but it allows to always make two read operations or two write operations in parallel, reducing the lookup, insertion and update delays.

To test the ability of each algorithm to avoid losing flows, a software simulation tool was built using the Python development language. A memory similar to the Combo SRAM memory (two times 1 048 576 72-bit slots) is simulated, as well as a timer that is incremented at each insertion request. Depending on the number of simultaneous flows in memory, the performance of each algorithm will differ. So to simulate 100 000 simultaneous flows, the delay for a flow to expire is set to 100 000. Then the algorithm under test is asked to store 100 000 random flows. These flows are only used to fill the memory, as it would be in normal use. The actual test is then made by first asking to store 100 000 new random flows, and then sending one lookup request for each flow to verify if the expected data is

returned.

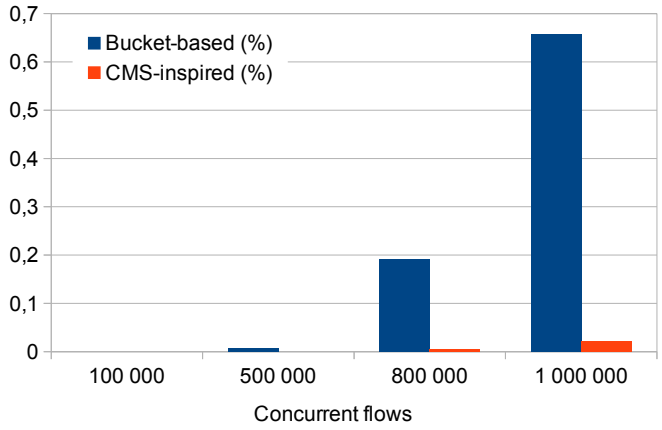


Figure 4.7: Percentage of lost flows depending on the number of concurrent flows

Figure 4.7 shows the percentage of lost flows for both the basic bucket-based algorithm and the CMS-inspired algorithm with a varying number of concurrent flows. With 100 000 concurrent flows (4.8% filling), no flows are lost by any algorithm. But for 1 000 000 concurrent flows (48% filling), 0.02% of the flows are lost by the CMS-inspired algorithm, and 0.66% by the bucket-based algorithm. So the bucket-based algorithm loses more than 30 times more flows than the CMS-inspired algorithm with 1 000 000 concurrent flows.

To check that supporting one million simultaneous flows is realistic, we compared this number to the traces described in Table 4.1. To adapt the bit rate, we supposed that if a trace at a bit rate of 1 Gb/s creates s simultaneous flows, it will, by aggregation of similar traffic, create $10 \times s$ simultaneous flows at 10 Gb/s. Using this technique, we found that at 10 Gb/s, the Ericsson trace would create about 29 535 000 simultaneous flows, the Brescia trace 161 000 and the ReSEl trace 418 000. The Ericsson trace generates many orders of magnitudes more flows, but it is the only trace captured at a very low data rate in a laboratory, so we consider that its traffic composition is not realistic. This leaves two traces under 500 000 simultaneous flows, which could be supported even at 20 Gb/s, the maximum bit rate of the Combo board.

So this method to store flows in RAM is realistic and scalable. Using a bigger memory would allow to store more simultaneous flows without making the algorithm more complex.

Hash functions

A hash function must be deterministic and spread its output values as uniformly as possible in the output range. That is to say that choosing a random input value, each value in the output range should have about the same probability of being returned by the hash function. Another important feature for storage algorithms is that a small change in the input value should change as much as possible the output value. This way, flows with the same IP addresses but different ports for

example should result in very different hash values, and be stored in different locations.

Many hash functions exist in the litterature and most have been implemented in hardware [DHV01, MCMM06, RLAM12]. But flow storage is only a tool for traffic classification on FPGA, and it must take as little space as possible so that the SVM implementation can be parallelized as much as possible. It must also be very fast, as eight hash functions will have to be computed before the packet can be tagged. Common hashes like Message Digest 5 (MD5) or Secure Hash Algorithm (SHA) have properties that are interesting for encryption algorithms, like the difficulty to reverse the functions. For flows storage, this is not needed. The only requirement is to avoid collsisions, as this would result in a storage error.

This is why we designed a very basic hashing algorithm, that is perfectly adapted to hardware implementation. The first step is to build the hash function:

- First the number of input bits I and output bits N are selected depending on the needs.
- Then a number of links L is selected. It can range from 1 to I.
- For each output bit, L different input bits are randomly selected.
- If an input bit is implied in no link, the random selection is started again, to make sure that all input bits impact the output value.
- The value of each output bit is the exclusive or of the L selected input bits.

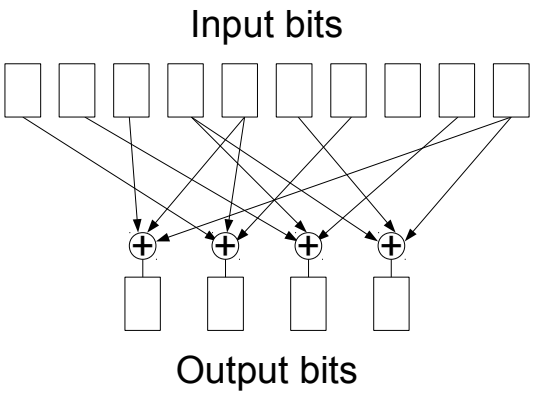


Figure 4.8: Example of hash function from 10 to 4 bits with 3 links

Figure 4.8 shows an example of hash function with $I = 10$, $N = 4$ and $L = 3$. With this function, the hash of 1010101010 would be 0110. In this example, the eighth input bit has no effect on the output. This shows that L must be chosen big enough so that all input bits have a high probability to have a noticeable effect on the output.

In the current implementation, the output of the hash function is on 32 bits. Only 21 bits are used as address by the CMS-inspired algorithm. The 11 others

are used to differentiate the stored values. The number of bits has been chosen to be 20. This means that 32 "exclusive or" between 20 bits are executed in parallel to compute one hash value. These parameters have been chosen experimentally to have a good accuracy while keeping the resources used on the FPGA low. The same parameters have been used for the software simulation.

Using these parameters, with 2 000 000 random input values, 436 collisions happened. Using the 32 least significant bits of the values from the MD5 algorithm, only 421 collisions happened with the same input values. The difference between these numbers is not significant, which shows that the algorithm can be efficient as a hash function.

It might be possible to improve the performance of this hashing algorithm by making a parallel with the theory on error correction codes, especially Low-Density Parity-Check (LDPC) codes. But the current version is enough to implement flow storage, and this is left for future work.

The main interest of this hash function is that it can be computed in only one clock cycle. This would be totally impossible to do with classical hash functions. All "exclusive or" operators are directly wired on the FPGA. So even if its properties are not as interesting as the MD5 or SHA algorithms for example, this hashing algorithm requires very low resources and is very fast. It makes it very interesting for flows storage on FPGA.

In the current implementation of the CMS-inspired algorithm, all eight hash functions are computed in parallel, adding only one clock cycle to the required delay for a flow lookup. This is crucial to make the classification in real time.

4.4.3 The RBF kernel

Once the flow builder works, the remaining task is the most complex: the SVM classification. We will first describe an implementation using the most common and generic kernel: the RBF kernel.

Computation of the RBF kernel function (Equation 4.4) requires three integer additions (one per vector component), three integer multiplications (to compute the squares), one multiplication by a floating-point constant, and one exponential computation.

Floating-point operations are complex to realize and use too much area on the FPGA. The best solution is to transform the algorithm to use a fixed-point model instead of a floating-point model. This section will present a direct hardware implementation of the classification algorithm using the RBF kernel, and the next section will present an implementation using an adaptation of this kernel, simpler to implement in hardware.

Operations

Multiplications are complex to realize in hardware. They could be done on specialized multipliers, but few are available on the chosen FPGA, so parallelism would be limited. They are required to compute the squares in the RBF kernel, but squares are symmetric functions with one integer parameter varying from -1500 to 1500 (maximum size of an Ethernet payload). A read-only memory

(ROM) with 1519 values is used to emulate squares. This memory is implemented on FPGA using specialized memory cells (BlockRAM).

To avoid the $y_{d,i} \times \alpha_{d,i} \times k_i$ multiplication, $ln\left(|y_{d,i} \times \alpha_{d,i}|\right)$ (ln is the natural logarithm) is precomputed, and the exponential used to get k_i is executed only after the addition of this term. Delaying the exponential computation transforms the multiplication into an addition. This way only one multiplication by a constant remains in the kernel computation, which is much simpler than a multiplication of two variables. This simplification has a cost, visible in Algorithm 1, as the exponential is moved from the computation of k (line 4) to the computation of S_d (line 7). So instead of computing the exponential once for all classes, it is computed once for each class. There are therefore eight times more exponentials to compute with our dataset. These computations are done in parallel, so more time is not required, but it does require more space on the FPGA.

A ROM is also used to emulate the exponential function. As the function $y = e^x$ tends towards 0 when x tends towards $-\infty$ and y is stored on a fixed number of bits, there is a minimum value x_{min} for which $y_{min} = e^{x_{min}} = 0$. The value x_{max} is determined experimentally by observing the input values of the exponential with different traces. With this technique and the quantization parameters described in section 4.4.3, only 10 792 values have to be stored in memory. To reduce this number even more, we use the property of the exponential $e^{a+b} = e^a \times e^b$: instead of storing values from -4 to 4, we store values from 0 to 4 and we store e^{-4}. This way, $e^{0.8}$ is read directly and $e^{-3.2}$ is computed as $e^{0.8} \times e^{-4}$. This technique costs one comparison and one multiplication by a constant, but divides the memory used by two. In the current implementation, the input domain is divided into eight segments, requiring seven comparisons and seven multiplications by a constant, and reducing the memory used to only 1 349 values. Multiplications by constants are converted during synthesis into simple shifts and additions.

Quantization

Table 4.4 shows the bit widths of different variables used in the SVM fixed-point model.

Variable	Integer part	Decimal part	Signed
Vector component	11	0	×
$\log_2(\gamma)$	3	0	×
$\ln(\alpha)$	4	10	✓
Exponential input	4	10	✓
Exponential output	12	7	×
Square input	12	0	✓
Square output	22	0	×
b	15	11	✓
S	15	11	✓

Table 4.4: Quantization of the main SVM fixed-point model values

These quantization parameters have been chosen so that the mathematical operations are carried out on values that are as small as possible, while keeping the same classification accuracy. Experiments show that the accuracy can be as good when using the fixed-point model as when using the floating-point model, but accuracy drops very fast if too small a parameter is chosen. Some parameters have quite large bit widths because the classifier should work whatever the SVM model. The possible values of the variables have been determined by analyzing SVM models learnt in different conditions.

The most critical parameters are the exponential and square input and output widths, because they change the width of the memories used, which may take a lot of space on the FPGA, decreasing the possibility of parallelizing the design.

11-bit-wide vector components have been chosen because we assume that the size of a packet will not be larger than 1500 bytes. This is the maximum size of standard Ethernet payloads. Jumbo frames are not supported. Supporting them would only require this width to be increased.

The γ parameter is chosen as a power of 2 to simplify hardware operations. Its value is determined for each trace by testing different values: first learning is applied on different subsets of the trace, then classification is applied on the whole trace. Results are compared to the ground truth and the parameter which gives the best accuracy is kept. With the traces we use, selected values are always higher than 2^{-7} and lower than 1, so the $-\log_2(\gamma)$ value is between 0 and 7.

The width of $\ln(\alpha)$ is chosen to fit the width of the exponential input, which is made as small as possible to decrease the memory used by the exponential. The square output is kept at 22 bits because it is then multiplied by gamma, which transforms up to seven integer bits to decimal bits. The b and S parameters share the same width because they are compared with each other.

Accuracy

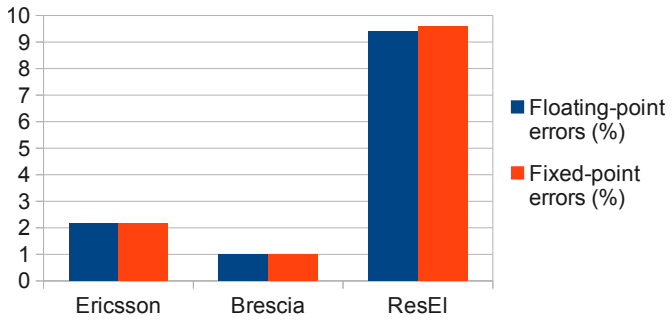


Figure 4.9: Differences between floating and fixed-point implementations for the RBF kernel (% of known flows)

To assess the loss in precision, a software implementation of the classification algorithm with the fixed-point model has been written. Figure 4.9 displays the percentage of errors compared to the ground truth for the floating and fixed-point implementations. The floating-point results have been described in Section 4.2.3.

There are actually fewer errors with fixed-point implementation for most traces. This very small improvement in accuracy is only due to errors which compensate by chance the SVM classification errors. This is not a reliable improvement and changes depending on the trace. But it shows that the transition to fixed-point does not decrease the accuracy of the algorithm. A relatively big percentage of differences is observed between the classification made by the floating and fixed-point implementations (between 0.32 and 7.6 %), but this is mainly on flows with no ground truth, so they cannot be considered errors.

4.4.4 The CORDIC algorithm

The RBF kernel is known to be efficient in most situations, and we have seen that it gives good results for traffic classification. But it makes it necessary to compute squares and exponentials, which is done using read-only memories. The routing of these memories increases the critical path on the FPGA, preventing it from working at high frequencies.

D. Anguita *et al.* suggested in an article [APRS06] a different kernel function that is much more adapted to hardware, because it can be computed using the CORDIC algorithm [And98]. This algorithm, described in Section 4.4.4, is used to approach trigonometric functions using only add and shift operations. These operations are the most simple arithmetical computations that a conventional digital circuit can handle.

Here is the suggested kernel function:

$$K(x_i, x_j) = 2^{-\gamma \|x_i - x_j\|_1} \tag{4.5}$$

It is very similar to the RBF kernel (equation 4.4), except that the exponential has been replaced by a power of 2 and the square function has been replaced by a simple absolute value. The replacement of the exponential by a power of 2 simply corresponds to a scaling of the γ parameter, but it makes some implementation simplifications more obvious.

Floating-point accuracy

The following section explains how using the CORDIC algorithm makes this kernel more adapted to hardware implementation. But before implementing it, the accuracy of the classification with this kernel must be checked. So we first integrated the floating-point version of the kernel in the LibSVM library to test the accuracy in software.

Results are shown on Figure 4.10. Using the floating-point kernel, classification errors represent 1.4 % of all known flows for the Ericsson trace, 0.75 % for the Brescia trace and 6.8 % for the ResEl trace. This is smaller than the 2.2 %, 1.0 % and 9.5 % with the RBF kernel (Figure 4.9). So this kernel is even more adapted to traffic classification than the RBF kernel. This is due to the use of a l^1-norm instead of the l^2-norm: it differentiates more vectors with multiple different components, that is to say flows that have more than one of the three packets with different sizes. These flows are more likely to be in different classes

than a flow with two packets of the same size, and one packet of a very different size.

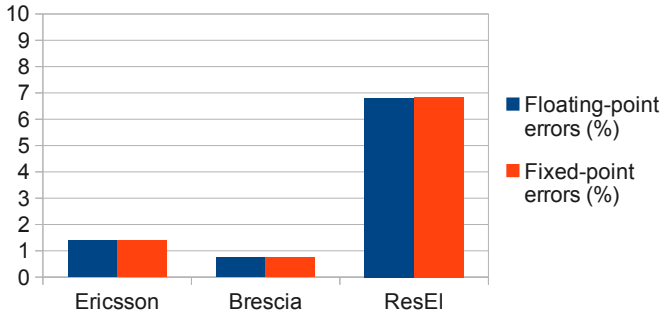


Figure 4.10: Differences between floating and fixed-point implementations for the CORDIC kernel (% of known flows)

Algorithm

The CORDIC algorithm is used to compute trigonometric functions using only addition and shift operators. It has also been extended to compute the exponential function, using the relation $e^x = \sinh(x) + \cosh(x)$. For the kernel computation, it will be used to get the value of Equation 4.5, combined with the multiplication by $\alpha_{d,i} \times k_i$ (line 7 of Algorithm 1). To do so, we will compute something of the form $B_0 2^{E_0}$, with input values B_0 and E_0 in $[0; 1[$ to ensure convergence. The way to get to this form from the equations is explained later in Section 4.4.4. The CORDIC algorithm works by defining two sequences:

$$E_0, B_0 = \text{initial values} \tag{4.6a}$$
$$B_n = B_{n-1}\left(1 + D_n 2^{-n}\right) \tag{4.6b}$$
$$E_n = E_{n-1} - \log_2\left(1 + D_n 2^{-n}\right) \tag{4.6c}$$

Each D_n is chosen equal to 0 or 1. The goal is to get E_n as near as possible to 0. $l_n = \log_2(1 + 2^{-n})$ is a positive, strictly decreasing sequence. If $l_n > E_{n-1}$, D_n is chosen equal to 0, and $E_n = E_{n-1}$, otherwise $D_n = 1$ and $E_n < E_{n-1}$.

The interesting thing about these equations is that they can be computed using only shift and addition operations once l_n has been pre-computed for each n. Indeed, at each step, $E_n = E_{n-1}$ and $B_n = B_{n-1}$, or $E_n = E_{n-1} - l_n$ and $B_n = B_{n-1} + B_{n-1} \times 2^{-n}$. Multiplying by 2^{-n} is equivalent to a right shift of n bits.

The way D_n is chosen, it is proved [And98] that $E_n \to 0$, so that with a big enough N, we have $E_N \approx 0$:

$$E_N = E_0 - \sum_{n=1}^{N-1} \log_2\left(1 + D_n 2^{-n}\right) \approx 0 \tag{4.7}$$

Using this approximate E_N value, we obtain for B_N:

$$B_N = B_0 \prod_{n=1}^{N-1} (1 + D_n 2^{-n}) \tag{4.8a}$$

$$= B_0 2^{\sum_{n=1}^{N-1} \log_2\left(1+D_n 2^{-n}\right)} \approx B_0 2^{E_0} \tag{4.8b}$$

It is proved in [APRS06] that if $E_0 < 1$ and $B_0 < 1$, in order to get an output accuracy of N bits (in fixed-point representation), N iterations of the algorithm are necessary, and some iterations have to be repeated to guarantee the convergence. For example, to guarantee an accuracy of 8 bits, $N = 8$ iterations are necessary and iterations $n = 2$, 4 and 8 have to be duplicated, so a total of 11 iterations have to be implemented. The way to choose which steps should be repeated depending on N is described in article [ABR03]. To avoid decreasing the accuracy, intermediate E_n and B_n results should be stored on $N + \log_2(N)$ bits, which means 11 bits in this example.

These accuracy values are guaranteed, but the number of steps and bits used in the implementation may be reduced by experimenting and testing that the resulting accuracy is still acceptable. This technique reduces strongly the area used by the implementation, but it may give poor results for some untested input values.

Implementation

Compared to Algorithm 1, using the CORDIC kernel changes the computation of k_i (line 4). But the $y_{d,i} \times \alpha_{d,i} \times k_i$ multiplication (line 7) is avoided by integrating $\alpha_{d,i} \times k_i$ into the computation of the power of 2 ($y_{d,i} = +-1$ so it is a simple inversion). So the CORDIC implementation described below changes line 7 of Algorithm 1, integrating the multiplication and the kernel function.

The CORDIC is used to compute $\alpha_{d,i} \times 2^{-\gamma \| x_i - x_j \|_1}$. As explained in Section 4.4.4, to be sure that the algorithm will converge, its input value E_0 must be in $[0;1[$, so $-\gamma \parallel x_i - x_j \parallel_1$ is separated into a negative integer part I and a decimal part D. Only D, in $[0;1[$, is used as input value E_0. A scaling of the α and b parameters by a fixed parameter is also computed beforehand to ensure that input value $B_0 = \alpha_{d,i}$ is always in $[0;1[$. This scaling by a fixed factor of the two elements of the inequation line 11 of Algorithm 1 does not change the classification results.

Algorithm 2 describes the CORDIC implementation. The loop is computed on the original N iterations, and the repeated steps. n represents the number of the iteration without repetition, m includes the repetitions.

As only the decimal part of $\gamma \parallel x_i - x_j \parallel_1$ has been used, the result must then be multiplied by 2^I, which is equivalent to a left shift of $-I$ bits (I is negative by construction).

Algorithm 2 CORDIC algorithm to compute $y_{d,i} \times \alpha_{d,i} \times 2^D$

$E_0 \leftarrow D$
$B_0 \leftarrow y_{d,i} \times \alpha_{d,i}$
$m \leftarrow 1$
for all iteration n from 1 to N (some iterations are duplicated) **do** {CORDIC loop}
  if $E_m \geq l_n$ **then**
    $E_m \leftarrow E_{m-1} - l_n$
    $B_m \leftarrow B_{m-1} - (B_{m-1} >> n)$
  else
    $E_m \leftarrow E_{m-1}$
    $B_m \leftarrow B_{m-1}$
  end if
  $m \leftarrow m + 1$
end for
The result is B_m

Quantization

To develop a fixed-point version of this new kernel, the same quantization process is used as for the RBF kernel in Section 4.4.3, in order to get as good a classification accuracy as with the floating-point model. The most important parameter to set is the number of steps in the CORDIC algorithm. In theory, the number of steps sets the number of bits of precision of the output. But by testing many options and comparing their accuracy, the complexity was reduced as much as possible and the parameters in Table 4.5 were selected.

Variable	Integer part	Decimal part	Signed
CORDIC output width	16	0	×
l_n	17	0	×
Scaled α	0	16	×
Scaled b	0	17	✓
Scaled S	8	16	✓

Table 4.5: Quantization of the main CORDIC values

The number of steps in the CORDIC is 15, and step 3 is repeated once. It is important to decrease this number of steps as much as possible because the CORDIC is implemented as a pipeline. This means that each step of the CORDIC is implemented on its own hardware, making it possible to start computing a new kernel value at each clock cycle. But this also means that each step has a cost in resources used on the FPGA.

Although the S parameter is only used to compare it to the b parameter, its integer part width is much bigger. This is important because S accumulates positive or negative values at each clock cycle, so it can grow and then decrease afterwards. If big values are cut, the final result will be changed.

Fixed-point accuracy

Figure 4.10 shows that, like for the RBF kernel, fixed-point implementation has the same accuracy as floating-point implementation. It is even a bit better on some traces, but this is not significant. A point not visible on this figure is that more differences are observed between the floating-point classification and the fixed-point classification than with the RBF kernel: for Ericsson there are 1.3% of flows classified in different classes instead of 0.92% and for Brescia 1.6% instead of 0.33%. The ResEl dataset is an exception with 0.96% of differences instead of 7.6%. As these flows in different classes are mainly flows for which we do not know the ground truth, they cannot be qualified as errors.

So the accuracy of SVM with the CORDIC kernel is a bit better overall than the accuracy of SVM with the RBF kernel for our traces.

An other unexpected difference is in the number of support vectors provided by the learning phase. Table 4.6 shows that models with the CORDIC kernel have almost half as many support vectors for some traces as with the RBF kernel. This side effect is very useful as the classification time of one flow depends linearly on the number of support vectors. But it should be checked for each trace, as nothing guarantees that the CORDIC kernel will always use fewer support vectors than the RBF kernel.

	Ericsson	Brescia	ResEl
RBF kernel	3 160	14 151	5 055
CORDIC kernel	1 745	8 007	4 838

Table 4.6: Number of support vectors of different SVM models

4.4.5 Comparing the two kernels

We have seen that in terms of accuracy, the CORDIC and RBF kernels are very similar, with a small advantage for the CORDIC kernel.

In terms of processing speed, the CORDIC models for some traces use almost half as many support vectors as the RBF models, although this is not so obvious for all traces. This means that the main loop has to be iterated less.

But the processing speed also depends on the possibility of duplicating the processing units, and on the maximum frequency at which the FPGA can work. So the next section will allow us to conclude which kernel is the fastest.

4.5 Performance of the hardware-accelerated traffic classifier

4.5.1 Synthesis results

Synthesis parameters

The proper behavior of the hardware implementation has been tested by checking that its results are exactly identical to the software implementation of the fixed-point model, first in simulation, and then implemented on a NetFPGA 1G and on a Combo board. So the classification results of the hardware implementation are those in Figures 4.9 and 4.10. This section focuses on performance in terms of the number of flows classified per second.

To assess the performance of the hardware implementation and compare it with the software implementation, it has been synthesized on a Xilinx Virtex-5 XC5VTX240, which is present on NetFPGA 10G boards. Only the SVM flow classifier is synthesized. The header parser and flow builder visible in Figure 4.2 are not included in the results. A different model has been tested for all 3 traces, using the RBF and the CORDIC kernel. The number of processing units has been changed as well to exploit the maximum parallelism on the FPGA while keeping a high working frequency. Table 4.7 presents the results of these syntheses for the Ericsson trace, Table 4.8 for the Brescia trace, and Table 4.9 for the ResEl trace. For each trace, the RBF kernel has been tested with 2, 4 or 8 computation units in parallel and the CORDIC kernel with 2, 4 or 16 computation units.

The number of occupied slices, registers and look-up tables (LUTs), as well as the maximum frequency, are given by the synthesis tool. They are an indication of the hardware complexity of the implementation. The number of cycles required per flow has been determined by analyzing the code of the hardware implementation. It increases with the number of support vectors in the model, and decreases with the number of parallel computation units.

Increasing the number of computation units makes it possible to classify a flow using fewer clock cycles, but it requires more resources on the FPGA. If the design contains too many computation units, it does not fit on the FPGA. So the synthesis fails or if it succeeds, the resulting working frequency is very low because routing the logic on the FPGA is too complicated. This is why results for the RBF kernel use only up to 8 computation units while up to 16 computation units are used for the CORDIC kernel: the RBF kernel uses more space on the FPGA, so the working frequencies with 16 computation units are too low to be usable.

Kernel comparison

Thanks to massive parallelism, hardware implementations all have better performance in terms of flows per second than software implementations. It can be observed that the RBF kernel uses more area on the FPGA in terms of slices than the CORDIC kernel. This is logical, as the CORDIC kernel has been specifically designed to be easily implemented on hardware. The RBF kernel uses a lot of

Trace	Ericsson					
Kernel	**RBF**			**CORDIC**		
Computation units	**2**	**4**	**8**	**2**	**4**	**16**
Occupied slices	6 223	11 362	22 431	4 767	8 092	24 325
Slice registers	8 395	21 010	44 302	8 966	17 168	59 347
Slice LUTs	19 451	36 345	79 602	14 880	24 973	74 664
FPGA usage (% of slices)	16.6	30.3	59.9	12.7	21.6	65.0
Maximum frequency (MHz)	153	165	171	201	199	183
Cycles per flow	1 602	814	421	903	469	146
Flows per second	95 636	203 091	405 894	223 090	424 148	1 254 223
Max. rate (Gb/s)	130	275	550	302	575	1 700

Table 4.7: Synthesis results of SVM traffic classification for the Ericsson trace on a Virtex-5 XC5VTX240 FPGA

Trace	Brescia					
Kernel	**RBF**			**CORDIC**		
Computation units	**2**	**4**	**8**	**2**	**4**	**16**
Occupied slices	11 007	15 716	24 311	6 131	9 108	25 444
Slice registers	7 362	17 644	38 644	8 048	16 462	60 945
Slice LUTs	38 242	51 914	83 968	20 440	29 137	80 762
FPGA usage (% of slices)	29.4	42.0	64.9	16.4	24.3	68.0
Maximum frequency (MHz)	126	71.0	77.6	176	201	172
Cycles per flow	7 098	3 562	1 795	4 034	2 034	537
Flows per second	17 707	19 944	43 256	43 704	98 822	320 075
Max. rate (Gb/s)	26.1	29.5	63.9	64.5	146	473

Table 4.8: Synthesis results of SVM traffic classification for the Brescia trace on a Virtex-5 XC5VTX240 FPGA

Trace	ResEl					
Kernel	**RBF**			**CORDIC**		
Computation units	**2**	**4**	**8**	**2**	**4**	**16**
Occupied slices	5 575	8 962	17 225	4 730	6 834	24 694
Slice registers	5 826	13 399	30 983	6 436	12 624	47 847
Slice LUTs	17 881	28 356	58 568	15 664	22 419	78 681
FPGA usage (% of slices)	14.9	23.9	46.0	12.6	18.2	66.0
Maximum frequency (MHz)	175	181	154	182	181	125
Cycles per flow	2 550	1 288	658	2 449	1 242	339
Flows per second	68 463	140 194	233 844	74 201	145 808	369 794
Max. rate (Gb/s)	75.6	155	258	82.0	161	408

Table 4.9: Synthesis results of SVM traffic classification for the ResEl trace on a Virtex-5 XC5VTX240 FPGA

memory as look-up tables, while the CORDIC kernel computes simple operations directly. The memory usage of the RBF kernel has a cost in terms of routing: the memory is not always located in the same place where computations occur, so long wires have to be used to fetch data from these memories. The delay induced by these wires lowers the maximum frequency at which the design can work. Tables 4.7, 4.8 and 4.9 show that CORDIC implementations work at higher frequencies than the RBF implementations.

Particularly with the RBF kernel, the Brescia trace gives poor results because of its low working frequency. The particularity of this trace is that the SVM model contains more support vectors than the others. They use too much space on the FPGA, which makes the delays caused by the look-up tables worse. Long and slow routes are created in the design and decrease its maximum frequency. The Ericsson and ResEl traces' SVM models have fewer support vectors.

Another important advantage of the CORDIC kernel is that models have fewer support vectors. This means that less memory has to be used on the FPGA to store these vectors, and less time has to be spent for classifying one flow. With n computation units, each support vector costs $1/n$ clock cycle.

Both the higher frequencies and lower number of support vectors give an important advantage to the CORDIC kernel, which is visible in the much higher bit rate supported for each trace. The area used is also smaller for the CORDIC kernel, which allows a higher level of parallelism.

Overall performance

The lowest supported bit rate is 26.1 Gb/s for the Brescia trace using the RBF kernel with 2 computation units. With the CORDIC kernel and at least four computation units, all traces can be handled with a data rate of more than

100 Gb/s. The Brescia trace is the most challenging: fewer flows can be classified per second by one computation unit because the model has the most support vectors. But parallelization makes it possible to reach high bit rates: with 16 computation units, the trace could be processed at 473 Gb/s.

It can be noticed that using two computation units, the CORDIC kernel is 70% faster in terms of flows for the ResEl trace than for the Brescia trace, but only 27% faster in terms of bit rate. This is because there are more flows to classify per second for the ResEl trace (see Table 4.2). Improvements compared to the software implementation are very important. For the Brescia trace, using the CORDIC kernel with 16 computation units, the bit rate is multiplied by more than 173 when compared to the software implementation (see Table 4.3), which is mostly due to the massive parallelism of the FPGA implementation.

One performance result that is not visible in the tables is the delay that the classifier adds to the packets if it is used directly on a network link to tag packets with their class number (and not just as a passive probe set up in derivation on the link). The current implementation sends the classification to the host computer instead of tagging the packets, but it could be modified without overhead. For now, 10 clock cycles are required between the arrival of the packet in the classifier and the time when the class of the packet is known. This delay is constant because a packet is considered unknown if the flow has not yet been classified (its class is not in RAM), so it does not depend on the classification time, but only on the flow reconstruction time. On the Combo board, packets are processed with a frequency of 187.5 MHz, which gives an induced delay of 53.3 ns. This figure does not include the time required to handle the Ethernet protocol, but it is a very acceptable delay as latency in IP networks is usually expressed in tens of milliseconds [ATT13]. The delay is also smaller than the minimum delay to receive a full Ethernet frame (67.2 ns), which guarantees that 10 Gb/s are supported without any need to parallelize the packet processing using a new pipeline.

To improve the supported speed for all traces, many directions are possible. Critical paths in the design may be improved to achieve higher frequencies, by adding registers that cut these paths. Algorithmic changes might also allow a reduction in the number of support vectors, as the use of the CORDIC kernel unexpectedly does. It is also possible to use more powerful FPGAs, or to use multiple FPGAs in parallel to reach higher speeds by having more parallel units. For example, we tested a synthesis using the CORDIC kernel on the Brescia trace with 32 computation units on a Virtex-7 FPGA, which corresponds to the Xilinx VC709 board. With this configuration, 661 987 flows per second are classified, a 107% improvement compared to the NetFPGA 10G. That is to say that the Brescia trace can be classified in real-time at a data rate up to 978 Gb/s.

Sensitivity analysis

Performance results presented above are valid with the chosen parameters of the SVM algorithm. Depending on the parameter, the effects on the computation complexity and supported speed can be more or less important. The most important parameters are:

- **The number of support vectors** affects the on-chip memory requirements and the computation time. Each computation unit is able to start computation for one new support vector at each clock cycle. So with n computation units, each support vector costs $1/n$ clock cycle.

- **The number of classes affects** the FPGA usage. With n classes, $n(n-1)/2$ binary classifications have to be made in parallel, so FPGA usage increases in n^2. For example, to support twice as many classes, four times less computation units could be implemented on the same FPGA, dividing the supported speed by four.

- **The number of flow features** affects the FPGA usage. Only the first stage of the kernel computation is affected by an increase in the number of flow features. For the CORDIC kernel, one subtraction and one addition is required for each feature. These operations are duplicated linearly when the number of flow features increases.

These parameters can all be changed in our generic SVM implementation simply by changing a variable value.

An implementation change that could also affect performance is the conversion of the ROMs used to store the SVM model into RAMs. This would allow to change the model without having to synthesize the design and program the board again. A small mechanism would have to be implemented to update the model, and the RAMs would have to be big enough to contain the biggest model supported by the board. So this would use a bit more space on the FPGA, but performance results should stay in the same range as the current implementation for the biggest model (Brescia).

4.5.2 Implementation validation

Experimental setup

Previous results were obtained using only a synthesis tool. In this section we present results obtained directly by sending traffic to a Combo card on which the classifier is implemented. Packets are received through an optical fiber at 10 Gb/s by the Combo card, which processes them, adds a tag with their class number in their header, and sends them to the computer hosting the card. A program runs on the computer and logs all received classifications. The traffic generator used for this task is a XenaCompact 10G generator [xen12].

Packet processing speed

The first thing to test is that the classifier handles traffic of 10 Gb/s without dropping packets. The most challenging packets to handle are the smallest ones, because handling a packet takes a constant time, whatever its size. The Ethernet standard defines smallest Ethernet frames as being 64 bytes long. Figure 4.11 shows the rate of 64-byte packets received by the classifier but not sent depending of the inter-packet time. The lowest time authorized by the Ethernet protocol is

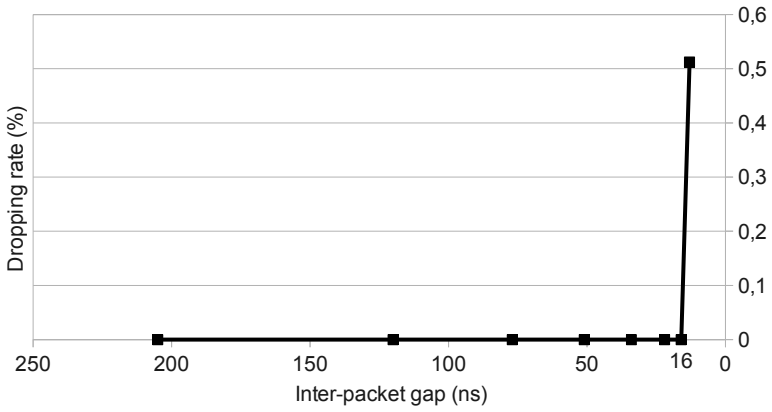


Figure 4.11: Packet dropping rate depending on the inter-packet gap

the time required to send 20 bytes, which is 16 ns at 10 Gb/s. This is precisely the last point for which the dropping rate is 0. The last point is for an inter-packet gap of 13 ns (the lowest possible gap with the generator) and some packets are dropped. So the classifier supports 10 Gb/s even with the smallest packets. This validates that the flow builder works fast enough to retrieve flow data for one packet in less time than is needed to receive a new packet header.

Flow processing speed

The second thing to test is the speed of the classifier itself. All the packets are handled properly, but how long does it take to classify one flow? To answer this question, 10 000 UDP flows of 6 packets of 64 bytes are sent to the classifier. The rest of the link is filled with ICMP packets that should be ignored. Six packets is the minimum to get one classified packet. Indeed, the first two packets are ignored, then the sizes of the 3 next packets are used for the classification. So the classifier starts working once the 5^{th} packet is received, and if it is fast enough, the class found is put in the 6^{th} packet and the next ones. Sending exactly six packets per flow, the maximum classification time is the delay between two packets. If the delay is too long, no packet from the flow is classified.

Figure 4.12 shows the percentage of classified flows depending on the packet rate for an SVM model with a CORDIC kernel, 1 000 support vectors and two computation units. Figure 4.13 shows the results with 2 000 support vectors. The packet rate is constant throughout each experiment. The clock used for the flow classifier is the same as that for the packet processing unit, so its frequency is 187.5 MHz on the Combo board.

In Figure 4.12 with 1 000 support vectors, the classification percentage is around 100% up to about 350 000 packets per second. In Figure 4.13 with 2 000 support vectors, it starts failing at about 180 000 packets per second. This is actually the maximum number of flows that can be classified per second, as the delay available to classify a flow is the delay between 2 packets. Using the same method as in Section 4.5.1, the theoretical maximum is 353 773 flows per second for 1 000 support vectors, and 182 039 flows per second for 2 000 support

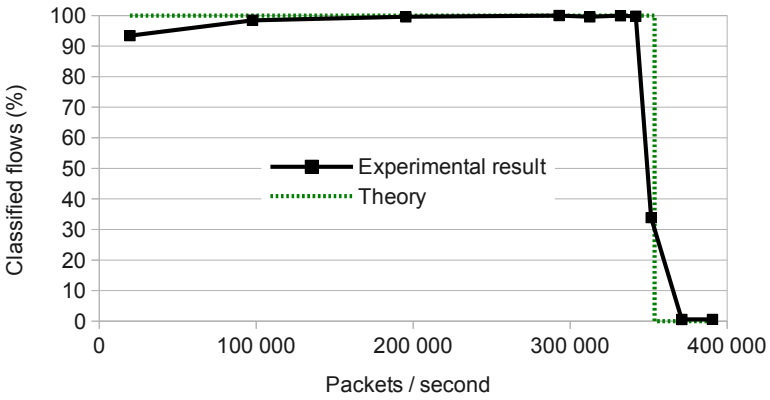


Figure 4.12: Classified flows depending on the packet rate for an SVM model with a CORDIC kernel, 1 000 support vectors and two computation units

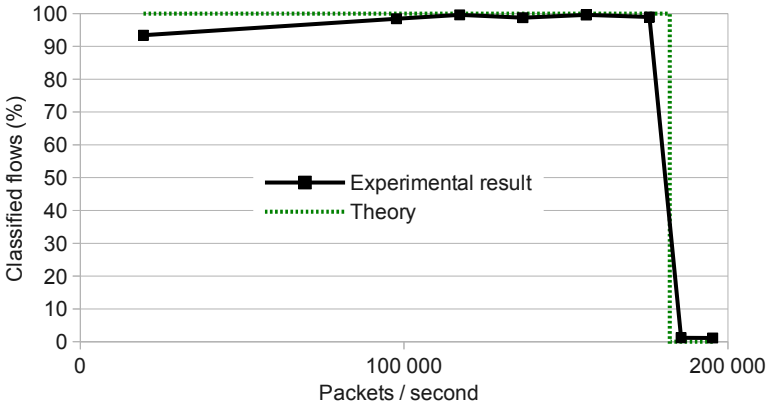


Figure 4.13: Classified flows depending on the packet rate for an SVM model with a CORDIC kernel, 2 000 support vectors and two computation units

vectors, as illustrated by the green dotted line on both figures. Differences near the maximum supported speed are only due to the small number of tested packet rates.

It can be seen that some flows are not classified even with a low number of packets per second. This seems to be due to the generator, which sometimes does not respect the configured inter-packet time. By logging sent packets in a file, we noticed that at low speeds, the generator tends to send packets in bursts, which means that the packet rate is not constant, so some flows are not classified when the packet rate is too high. This is a flaw of the traffic generator.

The concordance of the theoretical and measured results indicates that the implementation works as expected. It also validates the method used to compute theoretical performance values in Section 4.5.1.

4.6 Conclusion

This chapter describes the practical implementation of SVM-based traffic classification. Simpler classification algorithms like C4.5 exist, but we show that with the chosen flow features, that do not include source and destination ports, SVM provides a slightly better accuracy. The proposed SVM implementation is also very generic and can be adapted to other classification problems.

We use traffic-processing boards based on FPGAs like the NetFPGA 10G, taking advantage of very low-level access to network interfaces and massive parallelism.

We first build a simple but efficient flow storage algorithm inspired by the CMS algorithm. It supports with almost no loss one million simultaneous flows, and guarantees a constant update and look-up delay. It also uses very few ressources on the FPGA.

The main focus of this chapter is the flow classification process using the SVM algorithm, which is implemented in a fully parallel way thanks to a pipeline computing classification data on one support vector each clock cycle. To accelerate this process, multiple processing units can work in parallel, dividing the required time to handle flows faster.

Two different kernels were tested, namely the well-known and generic RBF kernel, and a kernel more adapted to hardware called the CORDIC kernel. They both give very similar classification accuracies, but the CORDIC implementation supports higher working frequencies and uses less area on the FPGA, which makes it possible to put more processing units in parallel. An unexpected improvement is that SVM models obtained with the CORDIC kernel have fewer support vectors than with the RBF kernel, which accelerates the processing.

Thanks to all these optimizations, flow classification can be performed at 320 075 flows per second for an SVM model with 8 007 support vectors, which would allow a real-time classification of the most realistic Brescia trace at 473 Gb/s.

To increase the speed supported by this traffic classifier, different parameters would have to be changed:

- Use higher-speed network interfaces

- Use a better FPGA to be able to parallelize the SVM algorithm even more. This way more flows can be classified each second.

- Use a bigger memory to be able to support more concurrent flows without increasing the risk of losing flow data.

Another possible improvement of the current implementation would be to make it more flexible by storing the SVM model in RAMs instead of ROMs, so that a new synthesis is not necessary at each model update.

This hardware classifier is another step towards flexible and accurate online traffic classification without subsampling on high bit rate links. The very efficient flow storage mechanism can be reused for other flow-based algorithms. The generic SVM implementation on hardware can also be used for other classification problems.

This traffic monitoring implementation on FPGA makes the advantages and drawbacks of this platform very visible:

- The development is complex and long. Problems like quantization have to be taken into account. Other problems that seem trivial in software, like flow storage, have to be handled manually due to more limited resources.

- Flexibility is not automatic. For example to allow dynamic SVM model changes, new developments would be needed.

- The speed improvement is huge, especially on algorithms like SVM that can be made massively parallel.

- It is easy to guarantee real-time processing, because each delay is known and no other task can interfere with the processing

Chapter 5

Hardware-accelerated test platform

We have seen two approaches to traffic monitoring. The first one to detect DDoS attacks was a full software implementation, very flexible and configurable. The second one to identify applications generating traffic was a full FPGA implementation, very efficient thanks to massively parallel computations. To prove the scalability of both solutions, we developed prototypes with 10 Gb/s interfaces. The ultimate goal is to support much higher speeds, but equipments supporting more than 10 Gb/s remain for now overly expensive for a prototype. Even with 10 Gb/s interfaces, the problem remains of how to check that the prototypes work the way they are expected to work. To do that, it is necessary to generate traffic with specified features at a configurable speed up to 10 Gb/s.

As we saw in Section 2.2.1, it is challenging to support a speed of 10 Gb/s in software, be it for receiving or sending traffic. Generating traffic at a specific speed on commodity hardware is so difficult, that many existing software generators have been shown to be unreliable [BDP10]. As commercial traffic generators are expensive, we will present a hardware-accelerated traffic generator that works on the Combo [IT13] FPGA platform, which we already own. This implementation will be an affordable and flexible solution for any researcher willing to try new traffic monitoring algorithms at high speeds. It will be possible to adapt to the widely available and affordable NetFPGA 10G [Net12] platform.

The presented traffic generator will have to be very reliable, especially in terms of generated data rate. The very low-level control provided by FPGAs will help for that. But it is also important for the traffic generator to be easily configurable, so as to generate different kinds of traffic depending on the algorithm under test. In Chapter 3, we preferred flexibility to performance, so as to provide a comprehensive and configurable monitoring platform to network operators. In Chapter 4, we focused on performance to support high data rates. For the traffic generator, a more balanced trade-off has to be found: a speed of 10 Gb/s must be supported without any problem, even generating small packets, but the profile of the generated traffic must be very easy and fast to configure. The way to add features to the traffic generator has also to be straightforward. As we think this generator can be useful to all researchers working on traffic processing, we make it fully open-source and available online [Gro13]. We hope others will help make it evolve with feedbacks and new features.

We will first list existing solutions for traffic generation. Second we will detail the specific requirements the traffic generator has to meet. Then we will describe the global architecture, divided into a software part, and a hardware part. Finally we will assess the performance of the generator by measuring generated traffic for specific use-cases with a simple commercial traffic analyzer [xen12].

5.1 State of the art on traffic generation

5.1.1 Traffic models

Traffic generation is the task of sending network traffic on a link to simulate the use of this link in an actual network of computers. It is used mainly by researchers to assess new algorithms performance, and by deployment teams to test new equipments.

Depending on the goal, the requirements on generated traffic can be very different. Here are some example use cases for traffic generation:

- To test the minimum supported data rate of an algorithm in the worst case, the generated traffic must have very specific features. If the algorithm processes packet headers one by one, it will have much more problems with small packets than with big packets, as the number of packets received per second will be much higher for the same data rate. If the algorithm considers only TCP traffic, other kinds of packets will be ignored, so a traffic with only TCP packets will be more challenging to support.

- To create plots of the performance of an algorithm depending on some varying traffic features, the composition of the generated traffic will have to be precisely specified too, and different traffic models will have to be tested.

- To assess the mean performance of an algorithm, realistic traffic will have to be generated. Only some generic features, like the data rate, have to be specified. The exact composition of the traffic (packet sizes, flow sizes, inter-arrival delays, packet data) only have to look realistic. The criteria to decide that a traffic is realistic depend both on the tested application, and on the location it will have on the network.

So the generated traffic may have to be either very specific to test edge cases, or realistic to test the normal behaviour of the application. Whatever the goal, two methods exist for traffic generation. The traffic can be fully synthetic, with each transmitted bit written using automata from a specification. Or the traffic can be replayed from a trace. In this case, a probe is used to save some actual traffic in a file, then the file is used to generate the exact same traffic.

The problem when replaying traces is that it is difficult to get real traffic data from operators. And a trace from a probe located in a certain network can rarely be considered as representative of all situations the algorithm should be tested for. It is even more complicated to find traces with some specific features, like the presence of attacks. So synthetic traffic provides more flexibility and allows a larger test cover.

Realistic traffic models

To generate synthetic traffic for edge cases, only some parameters have to be set (like the inter-frame delays, packet sizes or IP addresses), depending on the needs. But to generate realistic traffic, it is necessary to have a model of the real traffic. This is a very active and huge field of research. A 27-year old article presents a simple on/off model to characterize network traffic [JR86]. Packets are sent in multiple bursts separated by random time intervals.

Some years later, an article presents a more complex probabilistic model accounting for the burstiness of Ethernet traffic on aggregated links [LTWW94]. It presents the traffic as self-similar, that is to say that the shape of the traffic looks the same at different aggregation levels. This results in a burstiness of the traffic: intense transmission periods alternate with realitively idle periods. Self-similar stochastic models are used to characterize the traffic.

Instead of considering the traffic globally, a more recent article [SRB01] explains that only some big flows, called *alpha* traffic, are responsible for traffic burstiness and have a non-Gaussian behaviour. The remaining traffic is called *beta* and modeled as Fractional Gaussian noise. This model can be used to generate synthetic traffic using wavelets. Even more recently, [SMV10] provides a global probabilistic model of traffic on a backbone link based on fractional Brownian motion and fractional Gaussian noise.

These articles offer models that can be used to vary the data rate of a generator over time in a realistic way, so as to evaluate the average data rate an algorithm can support. But more accurate models can also be made when focusing on specific applications. A model exists for example [BAAZ10] for network traffic inside a data center. Traffic in a data center is generated by different servers communicating together, not directly by humans. It seems to result in an on/off behaviour, with some links alternating between busy and idle situations.

Some models are even more specific, like for example a model of traffic for wireless sensor networks [WA11]. This model takes into account the mobility of nodes, which communicate directly, and are not always able to reach the same neighbours. Sensors also need a strong energy efficiency, so they try to shorten their communication periods as much as possible. This results in a model far different from the one of an access network.

It can also be interesting to focus on a kind of traffic. For example, in [IP11], authors study only the web traffic, that is to say HTTP and HTTPS traffic. Another study [ZSGK09] is even more specific: its goal is to model Youtube traffic. These very accurate studies are more about measuring and analyzing the traffic than about creating a model to generate synthetic traffic. But the hints they provide about the way the traffic behaves could be used to build realistic models.

Our first goal is to test algorithms in the most demanding edge cases, so generating realistic traffic is not our main concern. But the traffic generator has to be reusable and extensible. This is why it remains important to build a flexible generator, that can be extended to take advantage of existing realistic traffic models. This way, the difficulty to implement a specific realistic model on the generator only depends on the complexity of the implementation of the

model (for example fractionan Brownian motion, fractional Gaussian noise or self-similar stochastic processes) on FPGA.

We detailed the different kinds of traffic models a generator may have to support. We will now see what existing generators offer.

5.1.2 Commercial generators

As traffic generators are widely used for Research and Development, many solutions exist. Different companies sell their own solutions, like Ixia [ixi12] or Xena [xen12]. These tools work out-of-the-box. Some software is usually provided with the machine to configure the parameters of the traffic to be generated using a simple interface. They are also often capable of receiving the traffic after it went through the device under test, and of providing statistics like the rate of dropped packets.

We recently acquired a XenaCompact traffic generator with two interfaces at 10 Gb/s. It is able to saturate both interfaces, even with the smallest packets. It is controlled remotely using a Windows program called XenaManager. Configuration of the generated traffic is done using the concept of stream. All packets in a stream use the same configuration. They can include counters that are incremented at each sent packet or random fields. The inter-frame delay can be set precisely. This generator is especially suited for edge case tests. It can generate packets with realistic payloads with existing protocols, but the inter-frame delay is constant between all packets of a same stream. Even if multiple streams can be intertwined, it does not offer a wide variety of traffic profiles.

An interest of the XenaCompact is that it also contains a very basic analyzer. It is able to measure the mean data rate, and to build histograms of the inter-frame delay of the received traffic.

The first drawback of commercial generators is that they are expensive. The way they are implemented is also kept secret, making it impossible to extend their functions when the configuration options are not enough. For example for the XenaCompact, it would not be possible to try to generate realistic traffic with random inter-frame delays following a custom probability density function.

5.1.3 Software-based generators

Any computer with a NIC is able to generate packets. This is why different traffic generators have been developed in software, to run on commodity hardware. Many focus on generating realistic traffic. For example, an implementation focuses on generating flows that have the same distribution of packet length as some well-known application types (HTTP, FTP, SMTP, etc.) [AOP+08]. The traffic composition is configured using a GUI and could be used to test the behavior of a device under some realistic traffic. It uses simple models (on/off for single applications and b-Model for aggregated traffic) to generate packets of realistic sizes. It does not support specifying the content of packets (header fields or payload). The maximum supported data rate is 1 Gb/s.

The Harpoon [SKB04] traffic generator goes even further in the idea of generating realistic traffic. It can generate TCP and UDP flows with a realistic distribution of packet sizes, flow sizes and inter-packet delays. The goal is to recreate the load of a real network on the device under test. Specialized traffic generators also exist, like a generator of online gaming traffic [SKS+10], based on a study of traffic generated by two popular online games. Such generators can be very helpful for specific use cases.

As discussed in the previous section, the problem for all these generators is to build traffic models. An interesting approach is the Swing traffic generator [VV09]. It uses a probe on an actual network to measure some features of the traffic, and is then able to generate traffic with the same features. It focuses on packet inter-arrival times and sizes, flow inter-arrival times and sizes, and destination addresses. Traffic is generated using a simulated network with the same characteristics (packet loss for example) as the measured network. So this generator is able to automatically create a simple model of the traffic, without any *a priori* knowledge of the network and applications.

These solutions focus more on the traffic model than on the ability to support high data rates. They are usually limited to traffic up to 1 Gb/s. An approach that could lead to support higher data rates is a distributed traffic generator [BDP12]. It uses multiple machines connected to the same network to generate traffic. Using this technique, it is possible to test the resistance of a wide network, with multiple links loaded simultaneously. It is also possible to generate traffic with higher bit rates. Although each machine is still limited to generate about 1 Gb/s of traffic, packets from multiple machines can be aggregated. But such a generator is complicated to setup only to test one equipment, and the generated traffic is not fully controlled, because it is difficult to synchronize multiple machines accurately. For example, it is not possible to guarantee the order of packets from two different machines after aggregation. Depending on the application, it may be a problem.

As explained in Section 2.2.1, standard operating systems do not permit to control accurately the delay between two sent packets. This results in inaccurate data rates and inter-packet delays in most software traffic generators [BDP10].

But some software traffic generators focus on the rate of the generated traffic and the reliability of announced inter-packet delays. They do not generate realistic traffic, but are great to stress test an equipment. To circumvent the problems due to the operating system, N. Bonelli et al. [BDPGP12a] implemented a solution that uses the custom network stack PF_DIRECT. With a powerful computer and a 10 Gb/s Intel NIC, they succeed to send packets at almost 10 Gb/s. But an use case shows that they reach the limits of what is currently possible using commodity hardware: they can send traffic at 10 Gb/s only if generated packets are big enough. With the smallest packets, only about 85 % of the link can be filled. This is an important drawback to stress-test an equipment, because small packets often constitute the most difficult traffic to handle. If the traffic generator does not support this traffic, it is not possible to guarantee that the algorithm under test will support it. The problem is that the CPU is not designed to handle the task of sending a huge number of small packets to the NIC. This is why

hardware-accelerated solutions are necessary.

5.1.4 Hardware-accelerated generators

As it is very challenging to obtain precise timing and high data rate in software, hardware acceleration is often required. In [BBCR06], a network processor is used to build a generator, which can send traffic up to 1 Gb/s. A limited number of flows with identical data are configured using a GUI and then sent at full speed. BRUNO [ADPF+08] is another approach, which uses the same strategy but supports an unlimited number of flows, because each generator instance can send a packet from any flow.

To get more flexibility than with a network processor, Caliper [GSG+12] uses a custom network processor implemented on the NetFPGA 1G. This implementation focuses on precise timing and works at 1 Gb/s. Scaling it to 10 Gb/s would be difficult because it relies on the computer to generate the traffic, so the computer would act as a bottleneck.

Using an FPGA with a network interface at 1 Gb/s too, A. Tockhorn et al. [TDT11] have a more hardware-focused approach: they stream headers with all data about one packet from the computer to the FPGA, which transforms this header into a full packet. As the header is smaller than the packet, the FPGA is able to reach a higher data rate than the computer. This approach would show its limits when trying to send small packets, as the header and packet would be of similar sizes.

FPGEN [SSG11] is a traffic generator based on an FPGA, which does not require a computer to stream traffic to it. It stores configuration in RAM, and can reach up to 5 Gb/s. It is focused on packets size and arrival time statistical distributions, and does not allow to specify some fields like IP addresses, ports.

An approach with goals similar to ours exists using the well-known NetFPGA 10G board [SAG+13]. Few details are given, but it is meant to be an open-source implementation using the four 10 Gb/s ports of the board. It includes a traffic analyzer for packets that are sent back to the generator. Implementation seems to focus on timestamping generated packets and analyzing the delays and latencies when they come back. It is configurable but does not advertise the ability to extend the hardware part of the generator easily. The described hardware architecture does not seem to make it easy to add features to generate traffic using complex traffic models.

In next section, we describe in details our needs for an affordable, configurable and extensible 10 Gb/s traffic generator.

5.2 An open-source FPGA traffic generator

5.2.1 Requirements

We have presented a scalable and flexible software-based DDoS attack detector in Chapter 3. We have also implemented a high-speed on line traffic classifier in Chapter 4. For both algorithms, prototypes have been developed that should

support a data rate up to 10 Gb/s. To be able to prove that this speed is actually supported, we had to send the most challenging possible traffic to these prototypes. We explained in last section that freely available generators do not fulfill our requirements, so we decided to build our own traffic generator.

Although the implementation was started for our own use, we realized that an affordable, configurable and extensible traffic generator, able to fill at least a 10 Gb/s link, can be useful for all researchers willing to test a network monitoring application at high speed.

The first purpose of this traffic generator is to push the limits of the device under test. This means that the most important is not to generate realistic traffic, but to generate traffic designed to be difficult to handle. The definition of such traffic depends on the functions of the device under test.

For example for the traffic classifier, two factors are important: the number of packets per second for flow reconstruction, and the number of flows per second for classification. Packets should be UDP or TCP because others are ignored. As some caching mechanisms are implemented, it is much less challenging to always send the same packet, than to send packets with changing flow identifiers. The payload of the UDP or TCP packets is of no interest to the classifier.

For the DDoS detector it is a bit different. The most challenging traffic is made exclusively of TCP SYN and ACK packets, because others are ignored. Packets with varying source and destination IP addresses simulate multiple attackers and multiple packets, although important numbers of packets should be sent to the same address so that an attack is detected.

Another interesting possibility for the DDoS detector would be to generate realistic traffic as background, and then generate one attack at a specified rate. It would allow to check the accuracy and detection delay of the algorithm. Contrary to other scenarios, this is not a stress test, and it requires the support of realistic traffic models. This test is not the most important for us, so we will first focus on generating synthetic traffic with specified parameters. But the traffic generator must be extensible, so that it is easy to add features, like the support of some realistic traffic models.

As we want researchers to be able to use the generator to stress-test all kinds of algorithms, the synthetic traffic has to be highly configurable: number of packets, size of the packets, content of the headers and payloads, etc. This configuration should be easy to do for anyone with basic network knowledge. The supported speed should be at least 10 Gb/s, and it should be easy to scale the implementation to higher data rates.

In terms of protocols, the generator should be based on Ethernet, as this is the most common low-level protocol and it is implemented on all platforms. But the protocols on top of Ethernet should be completely configurable: IPv4, IPv6, ICMP or others, and higher-level protocols UDP, TCP or even HTTP, SSH, etc should all be supported.

The traffic generator has to be open-source. It will be directly available on GitHub [Gro13]. We will keep maintaining and improving the current implementation. We also hope that others will use it. The more people use it, the more feedbacks we will get to improve the generator. Anyone is also encouraged to

develop new features and make them available to everyone.

5.2.2 Technical constraints

To allow a precise timing of packets, and to be able to guarantee that a 10 Gb/s link can be saturated, even generating the smallest packets, the traffic generator will be developed on an FPGA platform. The low-level control of the FPGA is great to guarantee the supported speed, and the massive parallelism enables complex processing of packets at high speed. Our preferred development platform would be the NetFPGA 10G, as it is affordable and the development platform is open-source. It also provides four interfaces at 10 Gb/s. But when we started the development of the generator, we only had a COMBO-LXT board, with a COMBOI 10G2 interface board offering two interfaces at 10 Gb/s. The board embeds a Xilinx Virtex 5 XC5VLX155T FPGA and comes with a development platform called NetCOPE [MK08]. The current implementation of the generator is for the Combo board. But porting it to the NetFPGA 10G platform should be fairly easy, as a full port of the NetCOPE platform on NetFPGA 10G already exists [KKZ+11], and we will do it as soon as possible.

Using an FPGA is efficient to support high speeds, but it does not favor flexible implementations. The generator should be both configurable and extensible. To make configuration as easy as possible, a control GUI is necessary. This is why we use an hybrid approach. The Combo board is embedded in a computer and can communicate with the CPU through a PCIe bus. So the generator is made of control software on the CPU and a generation module in the FPGA. The careful design of the generation module makes it easy to extend.

To make the traffic as configurable as possible, the most flexible way would be to specify each packet separately, using for example a trace file in the pcap format. But this would make it necessary to send configuration data to the FPGA for each packet. As explained is Section 2.2.1, this would create a bottleneck between the computer and the FPGA.

To avoid this problem, generated traffic is made of configurable streams. Streams are sets of packets that can be different but are generated from the same configuration. Differences in packets are due to the configuration indicating parts of the packets that are random or incrementing. The structure of the configuration will be detailed later. This approach is similar to [BBCR06, ADPF+08].

5.2.3 Global specifications

Respecting both the requirements and the technical constraints, we propose a traffic generator on a computer embedding a Combov2 FPGA board, able to saturate the two 10 Gbit/s Ethernet interfaces.

Configuration is made using a GUI. Generated traffic is made of a set of streams. To provide a maximum of flexibility, each stream is defined by:

a skeleton: list of bytes defining default packet data, including the full Ethernet frame but without preamble. This is the default packet that will be gener-

ated multiple times in a stream. All the packets of a stream start with the same skeleton and are modified afterwards.

a number of packets: this is the number of packets in a stream. Packets will be generated as fast as possible to saturate the link. The rate of the stream can be modified afterwards.

a list of modifiers: these are modules that modify packets before they are sent. Each modifier has a specific role and its behaviour can be configured. Modifiers make packets in a stream different from one another.

Modifiers are able to change the size of the packets, the time between each packet, or any bytes of the packets. This allows packets to vary by introducing, for example, incrementing fields, random fields or computed checksum fields. Configuration is able to activate and deactivate modifiers that are already present in hardware.

For instance, to define a stream of 50 ICMP packets that ping successively the IP addresses from 192.168.0.1 to 192.168.0.50, sent at 5 Gb/s, the configuration should be:

- Skeleton: the bytes of a valid ICMP/IP/Ethernet ping packet with destination address set to 192.168.0.1, IP header checksum bytes set to 0, Ethernet Frame Check Sequence (FCS) bytes set to 0.
- Number of packets: 50.
- Modifiers:
  - *Increment*: for byte 33 (least significant byte of the destination IP address), with a step of 1.
  - *Checksum*: for bytes 24 to 25 (IP header checksum), computed on bytes 14 to 33 (IP header).
  - *Ethernet FCS*: activated.
  - *Rate*: limit the rate to 5 Gb/s.

As shown on Figure 5.1, the *increment* modifier makes the destination IP address vary. The *checksum* modifier computes for each packet the checksum of the IP header, a redundant field made to check for transmission errors in the IP header. Otherwise packets would risk to be rejected as invalid because the checksum does not fit the modified header. The Ethernet FCS, another redundant field made to check for transmission errors in the whole frame, is automatically computed to make sure that the frame is valid. The *rate* modifier limits the rate to 5 Gb/s.

To make the streams totally configurable would require the hardware to contain a large number of modifiers of different types. For now, only the most commonly-used modifiers are available. But developing a new modifier is made as easy as FPGA development can be thanks to a flexible architecture. To add new modifiers to the hardware traffic generator, a new synthesis is necessary.

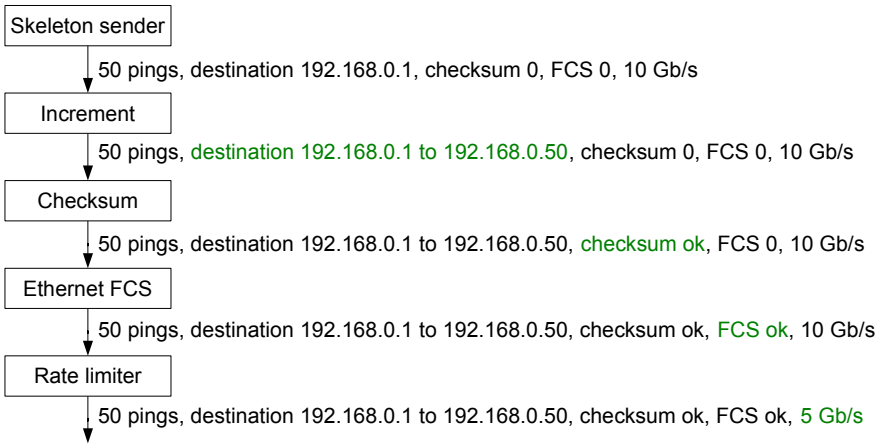


Figure 5.1: Example configuration to send ICMP ping packets

The generated traffic consists of configured streams mixed together. The data rate of each stream is modulated by modifiers. Ordering of packets in a stream is guaranteed, but no specific order is required for packets of different streams.

Next sections describe how this specification is implemented. First the software part is described, along with the communication between the CPU and the FPGA. Second the hardware architecture is detailed.

5.3 Software architecture

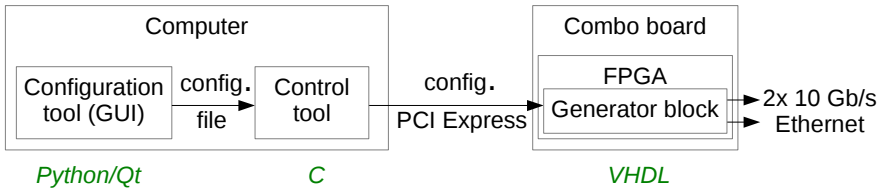


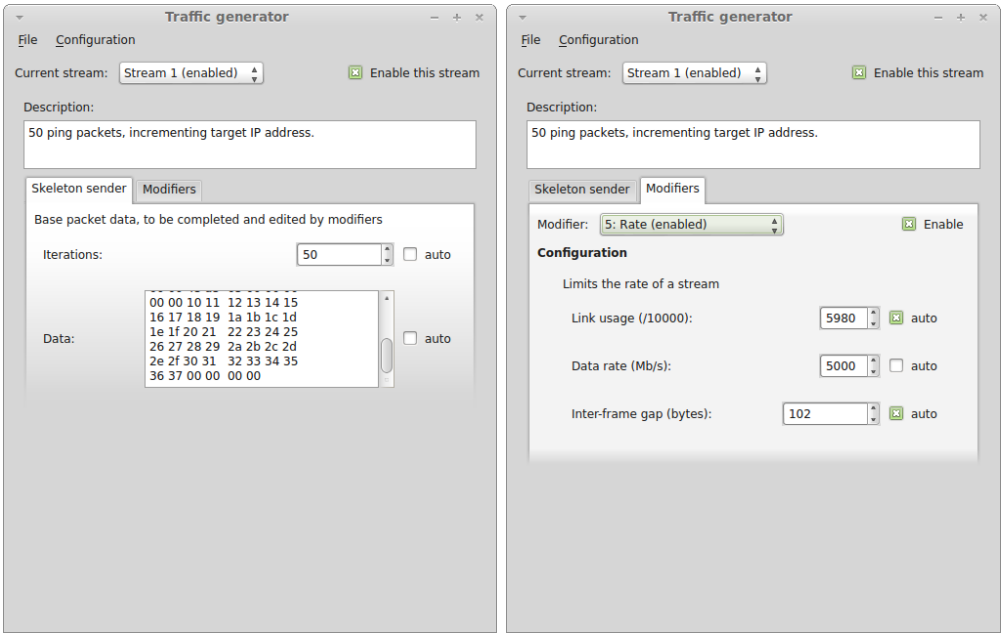
Figure 5.2: Global components of the generator

Figure 5.2 shows the global architecture of the generator. It is made of a computer embedding the Combo board with its two 10 Gb/s Ethernet interfaces. The configuration GUI on the computer is developed in Python using the Qt graphical framework. It produces a configuration file describing each flow. A small control tool developed in C is then used to send the configuration to the FPGA through a PCIe bus. The generator block on the FPGA is developed in VHDL. It receives the configuration and waits for an order from the control tool to start generating traffic.

All the C, Python and VHDL code is open-source and available online [Gro13]. A technical documentation to install, use or extend the traffic generator is also available in the same location.

5.3.1 The configuration interface

The generated traffic is made of streams. For each stream, the configuration is made of a skeleton, a number of packets and a list of modifiers. The skeleton and the number of packets actually are the configuration of a special mandatory modifier: the skeleton sender. So the configuration GUI has just to provide a way to configure modifiers for each stream.



(a) Skeleton sender (b) Rate modifier

Figure 5.3: Configuration tool

Figure 5.3 shows the interface of the configuration tool. 5.3a is the configuration of the *skeleton sender* and 5.3b is the configuration of the *rate* modifier. They are configured for the ping example of Figure 5.1. Configuration of each modifier is fully dynamic depending on user input. For example on the figure, the data rate is set by the user to 5000 Mb/s. The data rate is defined as the number of "useful" bytes of Ethernet frame, excluding the preamble and the inter-frame gap, sent each second. The link usage ratio and inter-frame gap are computed automatically from the rate. If the user changes the size of the skeleton, this will immediately affect the computed inter-frame gap, so as to keep the rate to 5000 Mb/s.

The architecture of the configuration tool is made to simplify the creation of new modifiers. The tool is separated into a GUI and a backend. The backend is responsible of listing configuration options for each available modifier, storing configuration values, and preserving the coherence of all configuration fields. The GUI simply uses the backend, presenting data in an user-friendly way. This makes the development of potential different configuration interfaces simple.

To list the possible configuration options, the backend needs to know what is available in the hardware generator block. This is done using a configuration file in JavaScript Object Notation (JSON) [Cro06], called *hardware.json*. It contains

the maximum number of streams, and an ordered list of modifiers available in each stream, in the order in which they modify packets. Each modifier has a type and an unique identifier. Different modifiers of the same type may be available, with a different identifier. Identifiers 0 and 255 are reserved and may not be used. As the identifier is coded on 8 bits, at most $2^8 - 2 = 254$ modifiers can be available for one stream. For now, we have never needed more than 6 active modifiers on a stream, so there is room for creativity.

Each modifier type has its own configuration fields, which work in a specific way. They may accept a range of integer values, a list of options, etc... Their value may also be automatically computed when other field values change. To make this flexibility possible, the configuration of each modifier type is described in a Python class. A typical modifier class contains:

A list of configurable fields. Generic field types like "unsigned" for integers, "select" for choices and "packet" for packet data are available. The modifier may list as many fields of these types as necessary. For each field, options are available, like a name and a description, and the possible range for an unsigned field. Two specific boolean options are also available: *editable* and *inConfig*. An *editable* field is visible in the GUI and can be modified by the user. An *inConfig* field will be stored as configuration data.

Event listeners. These are special functions that are called when a field value is changed directly by the user, or automatically. A modifier can listen to value changes for its fields, or for fields from other modifiers. These listeners are used to update fields automatically to keep the configuration integrity.

A "mandatory" flag. If the flag is set, it means that the modifier cannot be deactivated. For now, the only mandatory modifier is the skeleton sender. Figure 5.3a shows that the GUI displays mandatory modifiers in their own tab, instead of displaying them in the list of optional modifiers on Figure 5.3b.

This architecture makes creating a new modifier type very simple. The only thing to do is to write a Python class that describes the meaning of configuration data and the way it can be changed. There is no need to take care of the GUI or of the configuration storage.

Once the traffic is described properly, an option to export the configuration to a file is available in the configuration menu.

5.3.2 The configuration format

The format of the file exported by the configuration interface is designed to be easily sent to the hardware generator on the Combov2 board. It is a text file describing binary configuration data as hexadecimal values. The exact format is specific to the Combov2 board, so that it can be used with the hardware simulation tools provided with the board. This way, the exact behaviour of the hardware generator can be simulated. This is extremely useful to debug new modifiers.

The whole configuration is divided into frames representing the configuration for one stream. Each frame is divided into parts containing the configuration for one modifier. Each part is made of 64-bit words, this constraint is due to the 64-bit wide bus used on the Combo board to receive the configuration.

Data (64)		**Structure (4)**			
Id. (8)	*Config. (56)*	*SoF*	*EoF*	*SoP*	*EoP*
5	Rate	Yes		Yes	Yes
2	Ethernet FCS			Yes	Yes
3	Checksum			Yes	Yes
6	Increment			Yes	Yes
1	Skeleton (word 1)			Yes	
...					
Skeleton (word N)			Yes		Yes

Table 5.1: Sample configuration structure for a flow sender

Table 5.1 shows the configuration of the stream described in Figure 5.1. Four flags are used to signal a start of frame (SoF), end of frame (EoF), start of part (SoP) or end of part (EoP). As there is only one stream in this example, there is only one frame, with one part for each modifier. Parts start with an identifier, the same as in the *hardware.json* file, identifying the modifier affected by the part.

Each modifier is free to define the structure of configuration data inside its part, only the identifier is required, so that other modifiers ignore this part. Most parts are only 64 bits long because the SoP and EoP flags are set on the same data word. This leaves 56 bits for configuration data. But a part can be as long as necessary by using multiple 64-bit words. A part of 2 words leaves $56 + 64 = 120$ bits for configuration data, and so on. An example of long part is the configuration of the skeleton sender, because it contains all bytes of the initial packet skeleton.

The *Ethernet FCS* modifier is fully automatic and requires no configuration data. It just sets the FCS at the end of the frame. But it has still its own configuration part, using 64 bits in the configuration. Except the identifier, all other bits are unused and set to 0. The part is still necessary, because a modifier that receives no configuration is inactive and will not modify packets. The wasted bits are not really a problem because the configuration phase is not really time-constrained. Configuration data remain small and are sent almost instantly to the traffic generator. Traffic generation has not started in the configuration phase.

This is why the configuration structure is designed for simplicity, not for efficiency. Configuration is aligned on words of 64 bits to simplify development of new modifiers: reading aligned configuration data in hardware is simple. An example of modifier configuration will be detailed in Section 5.5.1.

5.3.3 The control tool

The control tool is a very small C program. It uses libraries provided by the NetCOPE platform to communicate with the Combo board. Its role is to send configuration data to the generator block, and to send orders to start or stop traffic generation. For now, it has no GUI and is used in command line. It could be integrated into the configuration GUI in the future.

The NetCOPE platform provides two communication methods between the computer and the FPGA. The first is for small data transfers: 32-bit registers on the FPGA can be made readable and/or writable from the computer like a memory. An address is assigned to each accessible register, and a simple C method call with the right address from the computer allows to read the value or to write a new value. The second communication method is for fast and big data transfers, it can be used to forward packets between the computer and the FPGA at link rate, that is to say 20 Gb/s (two network interfaces at 10 Gb/s each). It is accessible in C as two interfaces that can be read and written to in a similar way to network interfaces. On the FPGA side, data are sent and received through four 64-bit buses (one for each interface, and for each direction).

The first method is used to send orders to the FPGA. Two orders are defined: "start" and "reset". "start" orders the generator block to start sending packets. It must be sent after the configuration phase. "reset" stops traffic generation and orders the generator block to forget its configuration.

The second method is used to send configuration data to the hardware generator. The configuration file exported by the GUI is used. It is sent almost without conversion. One packet is sent for each modifier, with a special header to indicate the start and end of frames and parts.

5.4 Hardware architecture

The architecture of the generator block on the FPGA is of course more complicated to change than the software architecture. This is why it provides configuration mechanisms, able to change the behaviour of the generator without changing the hardware. But to create custom modifiers, it is still necessary to change the hardware. This is why the hardware architecture must be as simple and flexible as the software architecture.

Development on the Combo board is done in VHDL using the NetCOPE platform [MK08]. This platform intends to make development as independent as possible of the board architecture. The management of the Ethernet protocol is integrated. Received packets are made available through a specific bus type called FrameLink. Sending a packet simply consists in sending data on a FrameLink bus too. As two network interfaces are available, NetCOPE provides two buses for reception, and two buses for sending. Packets on the bus represent the whole Ethernet frame without preamble. Received packets start with a NetCOPE header that contains the size of the packet.

FrameLink is a one-way synchronous 64-bit wide bus developed specifically for the NetCOPE platform. It is able to transmit any kind of large streaming

data. It can be implemented easily on any FPGA. It includes two control signals that allow both the receiver and the sender to halt the data flow. These signals are asynchronous so as to avoid wasting time when the receiver is ready again. Data on the bus are divided into frames. Each frame is divided into parts. A part is made of multiple 64-bit words. The last word may not be fully used. The bus includes four specific control signals to specify the start and end of frames and parts. Each Ethernet frame is received as a FrameLink frame. The frame contains one part with the whole Ethernet frame data, as well as an optional part with the FrameLink header. The part with the FrameLink header is always included for packets received from a network interface, but should not be included when sending a packet to a network interface.

For the hardware generator block, incoming traffic is the configuration data sent by the software control tool. Outgoing traffic is the generated traffic. So configuration is received on an incoming FrameLink bus, and generated traffic is sent by an outgoing FrameLink bus. The software configuration file has been formated in a way that respects the FrameLink format, with a division into frames, parts and 64-bit words, so as to be easy to transmit on the bus.

5.4.1 Main components

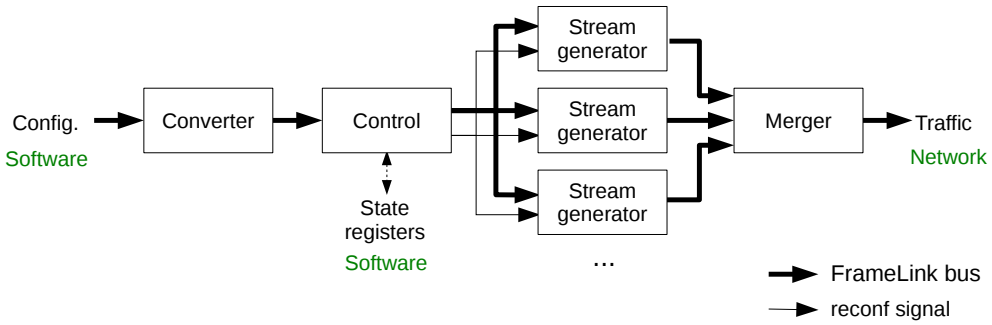


Figure 5.4: Architecture of the hardware generator block

Figure 5.4 details the design of the hardware generator block. It is structured around the FrameLink bus, used to transfer both the configuration and the generated traffic. It is made of different blocks described below.

The converter

This block is only necessary because of limits of the NetCOPE communication tools. It receives configuration data as sent by the software control tool. When received, the configuration of each modifier is in one different FrameLink frame.

The role of the converter is to transform the received configuration so that it respects the configuration format. In output, one FrameLink frame corresponds to the configuration of one traffic stream, and one FrameLink part corresponds to the configuration of one modifier.

The control block

The control block is in direct contact with the software control tool. It receives configuration data through the FrameLink bus, and it receives orders and sends status information through the state registers. The state registers are read and/or written by the software control tool.

The first role of the control tool is to receive configuration data and dispatch it to the stream generators. Each stream generator can receive at most one configuration frame. The first frame is sent to the first generator, and so on. The maximum number of supported simultaneous streams is thus limited by the number of stream generators implemented.

The second role of the control tool is to check the state of all stream generators. When stream generators have received their configuration frame, they stop accepting data on the FrameLink bus. This indicates to the controller that they are in configuration phase. When they are fully configured, they start accepting data on the FrameLink bus again. This means that they are ready to start generating traffic.

Data (64)		**Structure (4)**			
Id. (8)	*Config. (56)*	*SoF*	*EoF*	*SoP*	*EoP*
0	0	Yes	Yes	Yes	Yes

Table 5.2: Special configuration frame: *start*

The third role of the control tool is to send the start signal. When all the stream generators are ready, the control block checks periodically the state registers for a "start" order written by the software control tool. If it is present, the start signal can be sent. It actually is a special FrameLink frame sent to all stream generators at the same time. Its structure is presented in Table 5.2. It uses the reserved modifier identifier 0.

The last role of the control tool is to react to the "reset" order written by the software control tool in the state registers. When this order is read in any state, a reconfiguration signal called *reconf* is set to 1. This means that stream generators should stop generating traffic and get ready to receive a new configuration.

The stream generators

Each stream generator manages only one stream during a whole execution. It receives the configuration of the stream during the configuration phase, and then waits for the start word to start sending. If the *reconf* signal is set to 1 at any time, it goes back to the configuration phase.

Many stream generators are implemented in parallel so as to be able to handle simultaneously streams with different configurations. But each stream generator should be able to fill the 10 Gb/s link even with small packets, so that the link can be filled without losing control of the ordering of the packets.

Sending traffic at 10 Gb/s is not a problem for the FrameLink bus. It is 64-bit wide, and its frequency is the same as the whole design: 187.5 MHz, so the maximum data rate is 12 Gb/s.

The merger

The merger receives packets generated by all the streams and stores them in FIFO queues. It has one FIFO per stream generator. Each FIFO is checked successively in round-robin: if a packet is ready, it is sent, otherwise the next FIFO is checked.

The merger preserves the order of packets in each stream, because they are all stored in the same FIFO. But the relative order of packets from different streams cannot be guaranteed. The only guarantee is that each stream starts being generated at the same time, because the *start* command is sent to all the stream generators during the same clock cycle.

As all the streams are able to produce data up to 10 Gb/s, the total rate may be superior to the output link rate. In this situation, FIFOs get full and the merger starts refusing incoming data. Thanks to the control mechanisms of the FrameLink bus, this slows down all the stream generators. So packets are still generated properly, at wire speed. This situation can be avoided by configuring the *rate* modifier for each stream.

The architecture of Figure 5.4 is valid to send traffic on only one of the two possible network interfaces. It is also possible to send different traffic to both interfaces simultaneously. To do this, two mergers have to be used. Each merger receives traffic from one half of the stream generators. Each merger then sends traffic to a different network interface. So for example if ten stream generators are available, up to five streams will be generated per network interface. This allows the generator to send traffic up to 20 Gb/s.

5.4.2 Inside the stream generator

The stream generator is where the packets are actually created. An example of composition of a stream generator is visible in Figure 5.5. Each modifier has a unique identifier (number in green). The generator is a long pipeline where blocks are synchronized through the FrameLink bus. The interest is that all the blocks in the pipeline are constantly working: while the *skeleton sender* sends the last bytes of a packet, the *rate* block may already be working on the first bytes, wasting no clock cycle.

The *reconf* signal is received by all the modifiers. This is a kind of synchronous reset. Modifiers must stop sending traffic and forget their configuration when this signal is set to 1.

Blocks with a gray background in Figure 5.5 are mandatory in all streams. The *skeleton sender* is always the first modifier. It receives and stores the skeleton associated to its stream during the configuration phase. The skeleton is a piece of data from 64 to 1500 bytes, which contains the basic packet that will then be modified by the next blocks. This modifier also stores the number of packets that should be sent for this stream. Then when the *start* configuration word

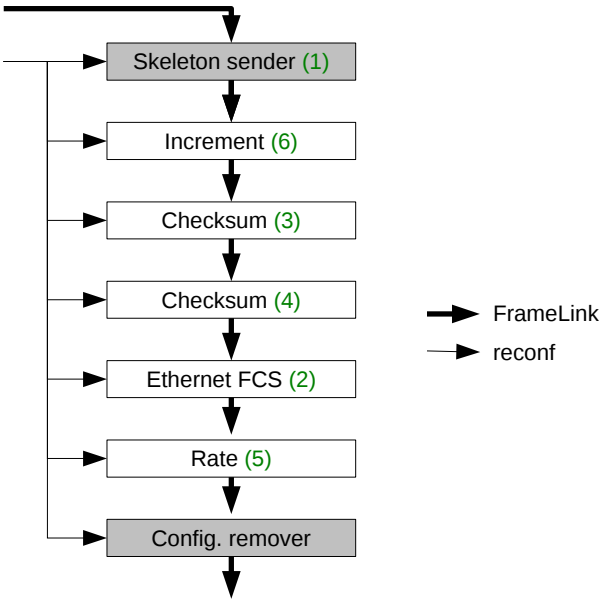


Figure 5.5: Example of blocks composing a stream generator

is received, the block starts sending the same skeleton continuously, until it has reached the number of packets to send. This is the only modifier receiving the *start* configuration word. The *skeleton sender* prepends a small header of two 64 bits words to each generated packet. Its format is specified by NetCOPE. It contains the size of the packet in bytes and in number of 64 bits words. This is useful to some modifiers that need to know the size of the packet before processing it.

The *config. remover* block is a very special modifier because it cannot be configured or deactivated, so it does not appear in the configuration GUI. It is always the last modifier of a stream. Its only role is to prepare generated data to be sent to the network interface, which consists in dropping the configuration frame and the small NetCOPE header prepended to each packet.

The ability to generate traffic at link rate depends on all the modifiers. As each modifier is able to stop data transfers on the bus, it can slow down the whole chain and prevent the stream to reach the link rate. The FrameLink bus works at 12 Gb/s, while the network interface supports 10 Gb/s on the Combo board. Moreover, a preamble and inter-frame gap of 20 bytes are added to packets to send them respecting the Ethernet protocol. This is why the 16 bytes of header prepended to each packet by the skeleton sender are not a problem. The bus is still much faster than the Ethernet link. The minimum useful speed for the FrameLink bus generating the smallest 64 bytes packets is 9.6 Gb/s, and only 7.6 Gb/s for the 10 Gb/s Ethernet link.

All currently available modifiers never slow down the bus. They may add some latency to packets because they use FIFOs. But this only delays the start of the traffic generation. It does not impact the data rate.

All white blocks in Figure 5.5 are optional modifiers. The functions of currently available modifiers are described below.

The *increment* modifier

It allows to modify a field of 2 bytes of the skeleton, setting it to the value of an internal counter that is incremented or decremented at each packet generated in the stream. The way the counter is incremented or decremented can be configured.

The configuration is made of:

- A field offset, indicating the number of bytes to count from the start of the packet to find the field to modify.
- A count mode: increment or decrement the counter.
- An increment value, indicating the positive value by which the counter should be incremented of decremented every time it is changed.
- A minimum and a maximum. The counter starts from the minimum or the maximum depending on the count mode. When the minimum or the maximum is reached, the counter is reset and starts again.
- A change skip period n, which means that the counter value is modified every $n + 1$ generated packet.

It can be used to make IP addresses vary for example. Combining two *increment* modifiers on the same stream, it is possible to increment a field of up to 32 bits, using the change skip period configuration.

On the FPGA of the Combo board, the maximum frequency of this module is 241 MHz and it uses 0.9 % of the space. The minimum frequency for the generator to work is 187.5 MHz, so this module will not slow down the processing. This module is very simple because it only has to select the right data and compute one addition.

The *checksum* modifier

It allows to modify one field of 2 bytes of the skeleton, setting it to the value of the checksum computed on a configurable range of bytes of the skeleton. It is compatible with checksums in IPv4, TCP and UDP headers. This may be used for example to set the checksum in the IP header after some fields of this header have been modified.

Its configuration is made of:

- A data start offset, indicating the number of bytes to count from the start of the packet to find the first byte that should be taken into account for checksum computation.
- A data end offset, indicating in the same way the last byte for checksum computation.
- A value offset, indicating the first of the two bytes that should be set to the checksum value.

- A pseudo-header mode, that can be none, IPv4 or IPv6. The IPv4 setting is used to compute a TCP checksum, which must be computed on an IP pseudo-header, made of some fields of the actual IP header. The IPv6 setting is not supported yet. It will be used to compute a checksum for TCP over IPv6.

- An IP header offset, indicating the start of the IP header. It is used only if the pseudo-header mode is not none.

The details of implementation of this modifier are described in Section 5.5.1, as an example of modifier design. On the FPGA of the Combo board, the maximum frequency of this module is 189 MHz and it uses 1.8 % of the space. It is slower and bigger than the *increment* modifier because the checksum computation is much more complex.

The *Ethernet FCS* modifier

It computes the FCS field of each generated packet and sets it, to conform to the Ethernet specification. The FCS field depends on all bytes of the frame. This block may only be omitted if the goal is to generate packets with wrong FCSs or if no change is made to the skeleton generated by the *skeleton sender*.

The FCS is always located in the last four bytes of the packet. The modifier locates these bytes using the size information available in the NetCOPE header of the packet. No specific configuration is needed. The modifier can simply be activated or not.

On the FPGA of the Combo board, the maximum frequency of this module is 204 MHz and it uses 1.3 % of the space. The module is simpler than the *checksum* modifier because it is less configurable, and an FCS is simpler to compute than a checksum.

The *rate* modifier

It allows to limit the data rate of this particular stream. If it is not present, the stream will be sent at maximum speed. It works by setting a constant inter-frame gap as an integer number of bytes on the FrameLink bus. As the FrameLink bus works at a speed of 12 Gb/s, one byte corresponds to 667 ps.

The configuration of this modifier is only made of the minimum average inter-frame gap in bytes on the bus. It is a minimum because the *rate* modifier is only able to slow down traffic on the bus. It cannot speed it up. So if other modifiers slow down the generated traffic, the actual average inter-frame gap may be wider than expected.

On the FPGA of the Combo board, the maximum frequency of this module is 194 MHz and it uses 0.6 % of the space. This module simply counts clock periods and stops the traffic if needed, which explains a very limited space usage.

Other modifiers

Modifiers described above are enough to test the DDoS detection and traffic classification algorithms under stress conditions. But many other modifiers may be developed to add features to the traffic generator.

An example of modifier for which development was started but is not finished yet, is a *payload* modifier. It is capable to append a random payload to generated packets. The payload size may be either fixed, or random. This allows to generate more variable traffic.

It would also be possible to develop modifiers to try and generate realistic traffic, using the models described in Section 5.1.1. Modifiers can act on the payload, the headers, the size of the packets, and the inter-packet delays to generate as realistic traffic as possible. It is possible to generate on/off traffic easily. More complex models can also be used, the development complexity depends only on the difficulty to implement the model on the FPGA, the control over generated packets is total.

5.5 Generator use cases

Users of the traffic generator can choose the customization level they need. At the deepest level, they can create custom modifiers to get the exact traffic features they need. Section 5.5.1 describes through an example how a new modifier can be created. If existing features are enough, it is possible to customize the composition of stream generators, duplicating useful modifiers, and fully removing useless ones. Section 5.5.2 presents the possibilities and limits to synthesize a generator block on FPGA. Finally, the generator can be used directly as a tool using the configuration GUI, with an already configured FPGA. Section 5.5.3 assesses the performance of the traffic generator as a simple tool configured using exclusively the GUI.

5.5.1 Design of a new modifier

Designing a new modifier is made in four steps:

1. The specification of the features of the modifier.
2. The specification of the configuration options and their hardware representation.
3. The description of the modifier in software for the GUI.
4. The development of the modifier in VHDL.

To illustrate the design of a modifier, we take as example the *checksum* modifier described in Section 5.4.2.

The role of the *checksum* modifier is to set checksums in IPv4, TCP or UDP headers after data changes. To do this, the modifier should read the proper fields, compute the checksum using the Cyclic Redundancy Check (CRC) algorithm, and

write it at the proper location in headers. The checksum value is always 16-bit long. The TCP and UDP protocols add a difficulty because a pseudo-header derived from the IP header has to be included to compute the CRC. So necessary configuration variables are:

- *start*: the byte offset at which computation should start (0 to 1517),
- *end*: the byte offset at which computation should end (0 to 1517),
- *value*: the byte offset at which the result should be inserted (0 to 1517),
- *ip*: if a pseudo-header has to be computed, the byte offset of the IP header (0 to 1517),
- *type*: a flag indicating if a pseudo-header should be computed, and if the IP version is 4 or 6 (none, 4 or 6).

The limit of all offsets at 1517 bytes is due to the Ethernet protocol, which states that an Ethernet frame can handle up to 1500 bytes of data (to which 18 bytes of header are added). The Ethernet protocol defines Jumbo frames, which can carry more than 1500 bytes of data, but they are not supported by most of the current modifiers. Supporting them would imply modifying the structure of the configuration of most modifiers to support offsets up to 9017. This change would be simple to make, but the need for generating jumbo frames did not arise yet.

Config. (56)					
start (11)	end (11)	value (11)	ip (11)	type (2)	0 (10)

Table 5.3: Checksum modifier configuration

Table 5.3 specifies the structure of the configuration part for the checksum modifier. It uses only one word on the FrameLink bus. One word is 64 bits, but 8 bits are used by the *identifier* from Table 5.1, so only 56 bits remain for the configuration. Any *identifier* that is not already used by another modifier can be chosen. For the *type field*, possible values are 0 for no pseudo-header, 1 for an IPv4 pseudo-header and 2 for an IPv6 pseudo-header. The last field is padding, made of ten bits at 0 to fill the 56 bits.

Once the configuration is specified, the best way is to write the Python class that describes the identifier for the GUI. The constructor of the class defines the name of the modifier and a short description. It also declares a list of the configuration fields. The order of the fields is the actual order of the bits in the configuration. For the checksum modifier, four fields of type *UnsignedField* are needed for the offsets. Their size is 11 bits. Then one field of type *SelectField* is used to describe the *type* field. It can take values “None”, “IPv4” or “IPv6”. The last field is a *BitsField* with a width of 10 bits and a flag *editable* set to false. It is only used as padding to fill the last bits of the configuration with zeros.

The Python class is registered as a modifier with the name *checksum*, and a flag *mandatory* set to false.

Now that the configuration is specified, the *checksum* block should be designed. The configuration phase is simple: wait for a data word with the *SoP* flag set and the first 8 bits representing the value of the modifier identifier. The value of the identifier is a parameter of the block. When this word is found, store all configuration data in registers. When data is received with the *EoF* flag set, go to the generation phase.

The generation phase can be divided into 2 steps:

- compute the checksum by adding interesting bytes according to the configuration at each clock cycle,
- set the checksum value at the proper position in the packet.

But the checksum is usually in packet headers, and it may be computed on bytes that are in the payload. So the packet has to be stopped until all the bytes have been received to compute the checksum. As processing speed is critical when generating packets, a pipeline has to be used in order to never stop the data flow during the process.

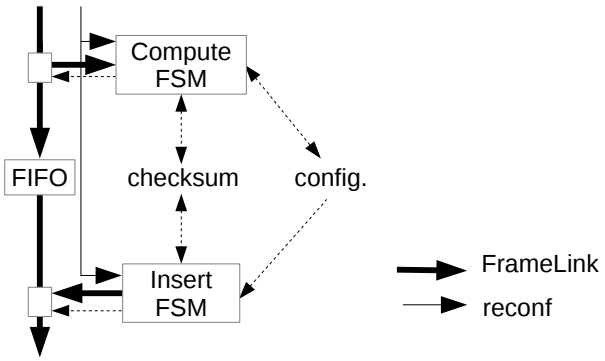


Figure 5.6: Architecture of the *checksum* modifier block

Figure 5.6 describes the architecture used for this block. Two Finite-State Machines (FSMs) are used. The *compute* FSM reads received data without ever modifying it. It manages configuration storage and checksum computation. The *insert* FSM lets data flow until it finds the location where the checksum should be inserted. Then it waits until the checksum is ready. During this time, received data is stored in the FIFO. When the checksum is ready, it sends it along with the received data, and lets data flow again until a new checksum has to be written in the next packet. The FIFO has to be big enough to store a whole packet, to be able to handle the case when the checksum is at the start of the packet, and is computed on the whole packet.

This process incurs a delay when the first packet is sent, but then while the *insert* FSM sends data stored in the FIFO, the *compute* FSM computes the next checksum. So the bit rate is not decreased by this block.

The source of this block is available [Gro13]. Note that although it is possible to specify in the configuration that the pseudo-header has to be created from an IPv6 packet, the function is not implemented yet and the *checksum* modifier supports UDP and TCP only over IPv4.

5.5.2 Synthesis on the FPGA

To use the traffic generator, the generator block first has to be synthesized. The generated bitfile is then used to configure the FPGA. We can provide a ready-to-use bitfile for the Combo board. But the synthesis allows to change some interesting parameters of the generator:

- The composition of a stream generator is described in a VHDL file. It is possible to add or remove as many modifiers as necessary. This is where newly designed modifiers must be inserted. Having a lot of *increment* modifiers chained may be useful to create traffic with highly varying headers for example.

- The number of stream generators is also a parameter in a VHDL file. No code has to be written to change this parameter. The value can simply be modified because the generator block is generic. For now, all stream generators are exactly the same. This is a design decision made to simplify the configuration. Of course, each stream generator receives its own configuration, but the same modifiers are available in each stream.

The ability to use multiple stream generators in parallel is important to generate diverse traffic, and to support multiple output interfaces. To explore the capacity in number of stream generators of the FPGA on the Combo board, we use the example stream generator of Figure 5.5 made of seven modifiers. We synthesize the full generator block, including the converter, control and merger blocks, on the FPGA of the Combo board.

Number of stream generators	1	5	2×3
Maximum frequency	189 MHz	188 MHz	189 MHz
Number of slice registers	2 560	12 978	16 606
Number of slice LUTs	4 321	20 735	25 572
Number of slices	1 539	7 904	9 679
Occupation	6.3 %	32 %	40 %

Table 5.4: Synthesis results of the sample traffic generator on the Xilinx Virtex 5 LX155T FPGA of the Combo board

Table 5.4 details the FPGA usage of the traffic generator with one stream generator for one interface, five stream generators for one interface, and three stream generators for each of the two interfaces. With only one stream generator, 6.3 % of the FPGA is used. So using multiple stream generators in parallel is not a problem. The maximum frequency supported by the design is over 187.5 MHz,

the frequency at which the Combo board works, so it can be implemented on the board without problems. Results in the table are for the traffic generator itself. They do not include the little space taken on the FPGA by the NetCOPE framework. So the results would be the same using the NetFPGA framework. Actually, as the NetFPGA 10G board embeds a more powerful FPGA, it provides more space to parallelize more stream generators.

With five stream generators, it is possible to send five concurrent streams on the same network interface to reach 10 Gb/s. With two times three stream generators, it is possible to send three concurrent streams on two network interfaces to reach 20 Gb/s. The choice of the number of stream generators and interfaces has to be made before the synthesis because the hardware architecture depends on the output interfaces. It would also be possible to have for example four stream generators for one interface and two for the other, but this possibility is not implemented to simplify the configuration.

Currently, the modifier using the most space on the FPGA is the *checksum* modifier. The computation it does is more complicated than other modules. Another reason that makes this module complex is that it can be configured to support checksums for different protocols. The maximum working frequency of the generator is also defined by the *checksum* modifier. This means that this module tries to make the most complex operation during one clock cycle. Simplifying this module would relax the constraints on the whole design, allowing to parallelize even more stream generators.

The synthesis and place-and-route steps needed to transform the VHDL code into a configuration file for the FPGA can take up to two hours. So putting more modifiers and streams than needed on the FPGA can be a good idea to avoid having to do it again when needs change. The GUI allows to activate or deactivate a modifier or a stream in one click.

5.5.3 Performance of the traffic generator

From the point of view of an end user using the traffic generator as a tool, modifier development and FPGA synthesis are rarely needed. The most important is to get good performance using the default traffic generator with the configuration GUI.

To make sure that the traffic generator sends traffic with the right features, an analyzer is needed. The XenaCompact commercial tool has some built-in traffic analysis functions. It is capable of building an histogram of the inter-frame gaps between packets. The inter-frame gap includes the Ethernet preamble. It is expressed in bytes. For a 10 Gb/s Ethernet link, one byte corresponds to 0.8 ns. The minimum gap enforced by the protocol is 20 bytes or 16 ns. From the measurement of the gap, the data rate can be computed. A limit of the XenaCompact is that it groups inter-frame gaps by four bytes to build the histogram. But the result is still accurate enough to have a good idea of the efficiency of the traffic generator.

Performance tests are made using the stream generator of Figure 5.5. Only one stream is activated. All modifiers are enabled to make sure that they do

not affect the output data rate. 100 000 frames of a fixed size are sent for each experiment.

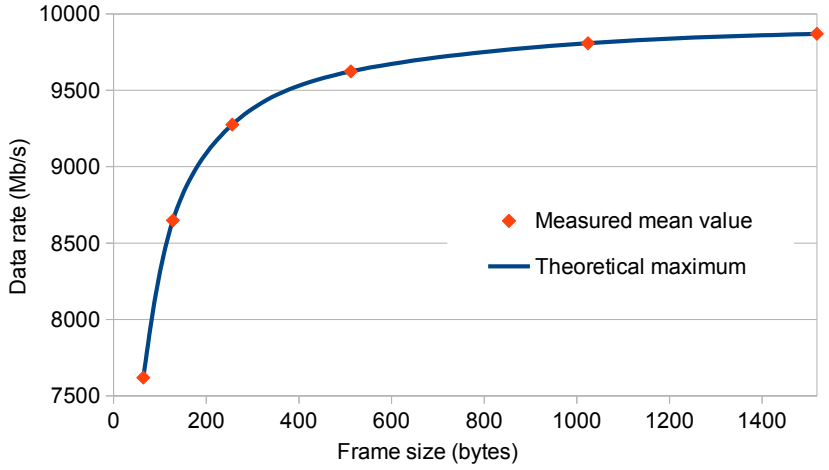


Figure 5.7: Generated data rate depending on the frame size

The first thing to check is that the traffic generator is able to fill the 10 Gb/s link for all packet sizes. To do that, the *rate* modifier is disabled. This way, the traffic generator sends traffic as fast as it can.

Figure 5.7 shows the results with frame sizes from 64 to 1518 bytes. The continuous line represents the theoretical maximum data rate. It is not 10 Gb/s because the Ethernet preamble and minimum gap are not considered as data. The maximum data rate is lower for small frames. The dots represent measured values. They are all perfectly on the line, meaning that the maximum data rate is supported by the traffic generator, whatever the frame size. The minimum frame size authorized by Ethernet, 64 bytes, is supported without problems.

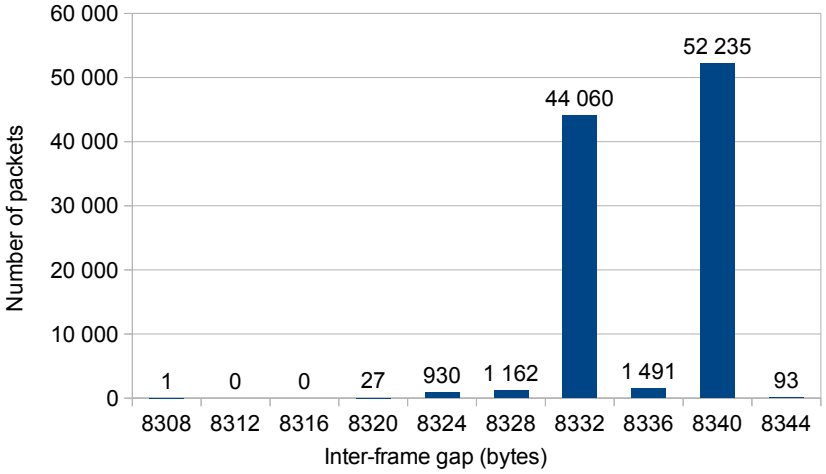


Figure 5.8: Repartition of inter-frame gaps for a configured gap of 8336 bytes

Once the maximum speed has been checked, it is also interesting to test the ability to send traffic at a specified speed, that is not the maximum speed. To do

that, the *rate* modifier has to be enabled. All other modifiers are still enabled. The skeleton sender is configured with frames of 64 bytes, and the rate modifier is configured with an inter-frame gap of 8 336 bytes, corresponding to a link usage of 1 % and a data rate of 76.2 Mb/s.

Figure 5.8 represents the repartition of inter-frame gaps measured by the XenaCompact. The mean inter-frame gap is 8 336.12 bytes, with a standard deviation of 4.20 bytes. The mean is very close to the configured value, and the deviation only amounts to 3.36 ns. So the *rate* modifier is an efficient way to set the data rate at a value lower than the link rate.

It can be seen that almost no packets are measured with the right inter-frame gap of 8 336 bytes, but two spikes exist at 8 332 and 8 340. This is due to the way the *rate* modifier works. It affects the inter-frame delay by stopping the FrameLink bus during a certain number of clock cycles. But eight bytes are sent at each clock cycle on the bus, so the *rate* modifier can modify the inter-frame gap by steps of eight bytes. To get the right mean inter-frame gap, it delays one frame a bit too much and the next a bit less, and so on.

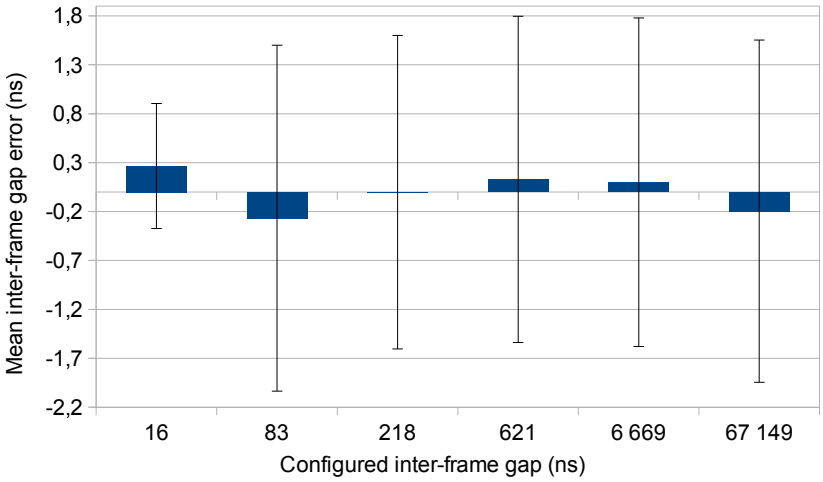


Figure 5.9: Inter-frame gap errors for a link usage from 100 % to 0.1 %

To make sure that the rate modifier behaves the same at any speed, the same experiment can be made while varying the configured link usage. The obtained histograms are very similar to the one of Figure 5.8. Some histograms have only one spike because the configured gap nearly corresponds to an integer number of FrameLink words.

Figure 5.9 presents the difference between the measured mean inter-frame gap and the configured inter-frame gap for a link usage varying from 100 % to 0.1 %. Vertical lines represent the standard deviation of the measured inter-frame gap. The mean error is always less than 0.3 ns, and the standard deviation is almost constant near 3.3 ns. The standard deviation is smaller at top speed, with a configured inter-frame gap of 16 ns, because the Ethernet protocol prevents the traffic generator from sending too fast.

It can be observed that it is better to disable the *rate* limiter, instead of configuring it with the minimum inter-frame gap, because it slows down the

generated traffic a bit. It is indeed more efficient to fill all FIFOs as fast as possible and let the Ethernet protocol manage these as fast as possible, than to send the traffic at the exact right speed.

The inaccuracies may be due to different factors: FIFOs delaying the traffic, or a way the included Ethernet IP transforms the traffic. Although this generator is less accurate than very expensive commercial solutions, it is also much more flexible and affordable.

5.6 Conclusion

This chapter presents an open-source traffic generator based on FPGA. Its use of the Combo board enables it to saturate its two 10 Gb/s interfaces easily, even with the smallest packets the Ethernet protocol allows. The modular architecture of the traffic generator is focused on one goal: be flexible at all levels.

The easiest way to customize the generated traffic is to use the configuration GUI. Some clicks are enough to specify the traffic in terms of streams of packets sharing certain features. Each stream can reach 10 Gb/s. Using multiple concurrent streams is useful to generate diverse traffic, as well as to send traffic to multiple network interfaces.

Contrary to commercial traffic generators, if the configuration GUI does not provide the proper options to generate the wanted traffic, the user can implement its own features. The generator is fully open-source and it is made to simplify the addition of new modules called modifiers. Developing a modifier requires knowledge of VHDL and a bit of Python, but it is made as simple as possible. All existing modifiers are documented and can be used as examples.

Although the generator currently works on the Combo board from Invea-Tech, it is very similar to the NetFPGA 10G, which is well-known and very affordable for academics. The NetCOPE platform used by the generator is compatible with the NetFPGA 10G, so porting the generator to the NetFPGA platform should be fairly easy. We intend to do so as soon as possible.

Currently, this traffic generator is focused on generating high-speed traffic for stress tests. It is not really adapted to generate realistic traffic. But this could be changed simply by developing new modifiers controlling frame sizes and inter-frame gaps to make them respect one of the numerous traffic models available in the literature.

As this traffic generator is an open-source project, the source code is available online [Gro13]. If some readers are interested in developing new modifiers, or want to help porting the generator to NetFPGA, or if they have ideas on how to make it better, they should not hesitate to get involved.

The architecture of this traffic generator is a good example of how the benefits of FPGAs can be used without hindering flexibility for the user. FPGAs bring strong benefits for real-time control. The low-level development makes controlling inter-frame gaps much easier than it would be on commodity hardware. Supporting the top data speed is done naturally thanks to the parallelism of the FPGA. Once the architecture is properly designed, the support is guaranteed.

But development on FPGA is complex and long, so it should be avoided as much as possible. Making the FPGA communicate with a computer to receive configuration data is a good way to implement a part of the task in software. From the point of view of the user, although FPGAs are low-level tools, they can be hidden behind user-friendly GUIs to be used by anyone. When FPGA development is needed anyway, it can be made simpler by defining clearly the architecture in which new blocks should be integrated, and by providing examples.

Chapter 6

Conclusion

6.1 Main contributions

This thesis is about finding the best methods to accelerate traffic monitoring to support high data rates. The best solution depends on the application, and on the most important requirements for the application. Acceleration has to be taken into account at the algorithmic level as well as at the implementation level to get the best results.

6.1.1 Development platform

The choice of a development platform determines the processing power available to the traffic monitoring application. Each development platform offers different advantages and drawbacks. I presented ten criteria that should be taken into account when selecting the right development platform:

The supported data rate. It is not simply the maximum data rate of the interfaces. On some platforms, it is very difficult to avoid dropping packets under stress conditions.

The computation power. It is important to implement the algorithmic part of the monitoring application. Some platforms offer a high level of parallelism, others offer a high number of operations per second or the ability to realize more complex operations. The best choice depends on the specificities of the algorithm: heavy computations and data dependencies for example.

The flexibility. A flexible application can be useful for network operators when facing changing requirements. Some platforms are more adapted to flexible applications, although flexibility can be achieved on all platforms with different degrees of effort.

The reliability. This is obviously an important requirement of network operators, although some applications may be less critical than others.

The security. Simple platforms can be guaranteed to be secure easier than complex platforms.

The platform openness. It may not be important for all applications, but the existence of an open community for a platform can make development and maintenance easier.

The development time. It determines an important part of the cost of an application.

The update simplicity. Applications deployed on a network have to be maintained, and some platforms make this easier than others.

The future scalability. Some platforms make it simple for developers to use the latest technology improvements and make their application more efficient over time.

The hardware cost. Monitoring applications can be made of a large number of probes, so the hardware cost can be a decisive factor.

I focused on four development platforms that provide acceleration for traffic monitoring. By studying their use in the literature, I was able to find their advantages and drawbacks:

- Commodity hardware with pure software development is very flexible and development is easy. The hardware cost is low although a powerful computer is required to support high data rates. The computation power is provided by the CPU. It can take advantage of algorithmic parallelism depending on the number of cores of the CPU. The problem is that an important part of the CPU is used for packet processing. The simple fact to receive packets is challenging on commodity hardware at high data rates, even with powerful NICs made to support up to 4×10 Gb/s. The reliability of the platform suffers from this difficulty to process packets.

- Powerful GPUs can be used on commodity hardware to provide more processing power. This does not change the packet processing issues, but more powerful algorithms can be implemented. The high level of parallelism offered by GPUs can make the implementation of some algorithms very efficient. But the communication between the GPU and the CPU can become a bottleneck depending on the application.

- Network processors can support very high data rates reliably and provide hardware-accelerated functions for common network monitoring tasks. The development is specialized for each NPU and cannot be reused on other platforms. Performance for an application highly depend on the model of NPU chosen.

- FPGAs can support high data rates. Development platforms are designed to guarantee the support of the interface rate without difficulties. They provide massive parallelism and a very low-level development. But this makes development difficult and long. As FPGAs are configured instead of programmed, they are also less flexible than other platforms.

There is no one-size-fits-all platform that would be the best for all traffic monitoring applications. The choice must be made depending on the most important requirements.

6.1.2 Software monitoring applied to security

The most widespread development platform is commodity hardware with pure software development. The interest is to use the flexibility, simplicity and low cost of normal computers. This is the solution I studied in the framework of the DEMONS European project.

I participated to the development of BlockMon, a flexible high-speed traffic monitoring framework. The interest of this framework is that it offers a very high degree of flexibility thanks to its architecture made of modular blocks that communicate through messages. The topology of the monitoring application, the traffic capture, the extracted data, everything can be customized. A GUI is even available to layout the organization of the blocks and the way they communicate. Blocks can be setup to be all on the same machine or distributed on multiple probes, without any new development.

But BlockMon is also focused on supporting high data rates. For this reason, the latest version of C++ is used for the development of blocks and messages. It allows a very careful management of the memory and the parallel processing. Everything is done to avoid copying data in memory, because it takes time. Each block can be configured to work on a specific thread on a specific core of the CPU. To process packets, a block allows the use of the PFQ network stack, a stack that is optimized to support traffic over 10 Gb/s.

Inside BlockMon, my first contribution was to develop reusable libraries and blocks for basic tasks that may be needed for traffic monitoring, like maintaining a large number of counters in a memory-efficient way, or detecting abrupt changes in series. Algorithms used for these tasks are called CMS [CM05] for counting and CUSUM [TRBK06] for change detection. They are examples of algorithms that are adapted to an efficient implementation to support high data rates. CMS is adapted because it uses little space and requires a processing for each packet that takes a small constant time. CUSUM is adapted because it has optimality properties that guarantee a good accuracy. It requires a long processing, but it is not done for each packet, but only periodically.

My second contribution was to test the flexibility and performance of BlockMon by focusing on a specific use-case in the domain of security: the detection of a type of DDoS attacks called TCP SYN flooding attacks. I used the CMS and CUSUM algorithms to build a fully modular TCP SYN flooding detector, able to raise alerts indicating the IP addresses of attackers and victims. Alerts are exported in the standard IDMEF format. Thanks to BlockMon, it is possible to use the application on a standalone machine, or to use multiple probes distributed on the network and a central collector. Configuration is simple using the GUI.

To test the supported data rate of this monitoring application, I used a computer with eight cores on two CPUs and an Intel 2×10 Gb/s NIC. The PFQ network stack exploits the NIC as well as possible. Results show that the multi-

threading offered by BlockMon is extremely important for monitoring applications. The use of BlockMon does not slow down the application, which is able to support a 10 Gb/s traffic if received packets are not too small. This limitation is due to a communication bottleneck between the NIC and the CPU, not to BlockMon.

To obtain good performance results, I had to fine-tune a large number of settings on the NIC and the computer. This task is time-consuming and does not even allow the test machine to support a 10 Gb/s traffic in the worst case, even when simply counting packets. This outlines the limits of traffic monitoring on commodity hardware: supporting 10 Gb/s or more is very challenging and a perfect result is difficult to guarantee. If the application has to support even higher data rates, the only solution is to wait for technological improvements that will make computers more powerful, or to use a hardware-accelerated development platform.

This implementation shows the main advantage of commodity hardware: the flexibility of the resulting application. A GUI allows to change from a single-node to a distributed application in some clicks. But it also shows the limits in terms of supported data rate. Simply receiving packets without dropping them is challenging. In terms of computation, the lightweight DDoS detection algorithm can support normal traffic up to 10 Gb/s, but an heavier algorithm would slow down packet processing. The development speed on commodity hardware is a bit lower than expected because software has to be fine-tuned to get good results, and a long test process has to be done to select exactly the right configuration to get the best performance from a given computer and NIC.

Contributions on BlockMon and software traffic monitoring have led to this publication:

- Andrea di Pietro, Felipe Huici, Nicola Bonelli, Brian Trammell, Petr Kastovsky, Tristan Groléat, Sandrine Vaton, and Maurizio Dusi. Toward composable network traffic measurement. In *INFOCOM 2013: 32nd IEEE Conference on Computer Communications*.

6.1.3 Hardware monitoring applied to traffic classification

As software traffic monitoring had shown its limits, I decided to test hardware traffic monitoring. I used the FPGA platform COMBO from INVEA-TECH because it offers the most freedom to explore new architectures. I focused on an use case that requires heavier computations than DDoS detection: real-time traffic classification. The classification is based on very simple flow features: the size of the first packets of the flow.

My first contribution was to show that using these features, a learning algorithm called SVM can give better results than other widely-used algorithms like C4.5. Although SVM is more complex than C4.5 to implement, it is an interesting challenge because SVM is widely used for diverse classification applications.

Before implementing the classification algorithm itself, I had to find an efficient flow storage algorithm, using a small memory space to store data about a very large number of flows. The algorithm also had to guarantee small update

and look-up delays because these operations happen for each received packet. I suggested a new solution inspired by the CMS algorithm, and showed using simulation that it is able to store data about more simultaneous flows than existing solutions [Mar08]. I then implemented the algorithm on FPGA, proving that it requires only a small space on the FPGA.

To make the classification algorithm more adapted to an hardware implementation, I tested an existing variant of SVM that uses a different kernel function. This function has to be computed a lot for classification. The variant uses an algorithm often used to compute trigonometric functions in hardware: the CORDIC. I showed that this variant was very adapted to traffic classification, causing no loss in terms of classification accuracy.

I implemented both the classical version of SVM and the variant on FPGA, exploiting as much as possible the massive parallelism. For simplicity, I called the classical version "RBF", from the name of the classical kernel function, and the variant "CORDIC". I showed that the CORDIC version was more efficient, processing more flows per second because it can be parallelized more efficiently than the RBF version. Actual results obtained using a traffic generator showed that the classifier supports 10 Gb/s without problems. The limit is more in the number of flows the classifier can process each second. But this number is still massively better using an FPGA than in pure software.

This implementation shows that FPGAs make it easy to process packets at high data rates. They also provide massive improvements in the speed of algorithms that can be parallelized like the SVM algorithm. Finding variants of algorithms that are more adapted to hardware can be a great way to improve performance. But this implementation also shows the drawback of FPGAs: the implementation is not very flexible, and changing the parameters can take a long time.

Contributions on the hardware implementation of the traffic classifier have led to these publications:

- Tristan Groléat, Matthieu Arzel, and Sandrine Vaton. Hardware acceleration of SVM-based traffic classification on FPGA. In *3rd International Workshop on TRaffic Analysis and Characterization* co-located with *Wireless Communications and Mobile Computing Conference (IWCMC)*. Limassol, Cyprus, 2012.

- Tristan Groléat, Matthieu Arzel, and Sandrine Vaton. Stretching the edges of SVM traffic classification with FPGA acceleration. **Final review process** in *IEEE Transactions on Network and Service Management.*

- Tristan Groléat, Sandrine Vaton, and Matthieu Arzel. High-Speed Flow-Based Classification on FPGA. In *International Journal of Network Management.*

6.1.4 Hardware-accelerated test platform

To test the traffic classifier and the DDoS detector, I needed a traffic generator. The task of generating traffic is very similar to the task of monitoring it. The same

problems arise when trying to support high data rates. A traffic generator that cannot reliably fill a 10 Gb/s link is useless: no reliable results can be obtained by testing an algorithm with a generator that is unreliable. For this reason, I decided to use the same platform as for the traffic classifier: the Combo board with an embedded FPGA.

My contribution is a fully open-source, flexible and extensible traffic generator able to fill two 10 Gb/s links simultaneously, even with the smallest Ethernet packets. The development of the generator was started with the help of Télécom Bretagne students in the framework of two academic projects.

The modular hardware architecture of the generator, made of multiple parallel pipelines, guarantees that it can support the interfaces data rate if no mistake is made in the design of the blocks that compose the pipelines. The generated traffic is divided into flows. Each flow can reach a data rate of 10 Gb/s. Each flow is customized by multiple chained modifiers. Each modifier is able to change the characteristics of generated packets: delay, size, header and data. This method guarantees a good flexibility to define the traffic.

A GUI is provided to configure the generated traffic easily. It addresses the problem encountered with the traffic classifier: FPGAs are not very flexible. For the traffic generator, flexibility is a requirement from the start, and the whole architecture has been thought to be flexible. The use of a custom software program to configure the hardware traffic generator brings a part of the flexibility of software to the traffic generator.

If the GUI does not provide the options required for a test, it is possible to develop new modifiers that can be added simply to the generator. This way, the features of the generator can be indefinitely extended. It would for example be possible to generate realistic traffic with the generator by developing new modifiers.

Using a commercial analyzer, I proved that the traffic generator generates traffic at 10 Gb/s easily, and that it can be configured to generate traffic reliably at a lower speed too. Currently, the generator is implemented on a Combo board, but it could easily be ported to the NetFPGA board. It would then be able to fill simultaneously four links at 10 Gb/s.

I think this traffic generator can be useful to other researchers, and I hope some will be interested in extending its features to test different applications. This implementation shows that applications on FPGA can be flexible if this requirement is taken into account in the design. Combining the flexibility of software with the performance of FPGAs is possible. This is done here by letting software configure the FPGA. This way, a simple GUI is available for configuration. The architecture of the hardware traffic generator is designed for flexibility too because it allows adding new modules easily. This process requires knowledge about FPGA development and is still slower than pure software development, but it does not require a full knowledge of the architecture of the generator.

Contributions on the open-source traffic generator have led to these publications:

- Tristan Groléat, Matthieu Arzel, Sandrine Vaton, Alban Bourge, Yannick Le Balch, Hicham Bougdal, and Manuel Aranaz Padron. Flexible, extensi-

ble, open-source and affordable FPGA-based traffic generator. In *Proceedings of the first edition workshop on High Performance and Programmable Networking (ACM HPPN)*. New-York, USA, 2013.

- Tristan Groléat, Sandrine Vaton, and Matthieu Arzel. Accélération matérielle pour le traitement de trafic sur FPGA. In *15èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (AlgoTel)*. Pornic, France, 2013.

6.2 Acceleration solutions comparison

Based on the different implementations of traffic monitoring applications I have worked on, I can provide some tips on the development platform to choose depending on the requirements:

Supported data rate

- The supported data rate is the weak point of pure software implementations. Although it is possible to support a data rate of 10 Gb/s or even more on commodity hardware, stress traffic with particularly small packets may still cause some packets to be dropped, even with a very light application like a packet counter, as seen in Section 3.5.2. So if you need to guarantee a perfectly supported data rate above 10 Gb/s, a pure software implementation is not adapted.
- The use of a powerful GPU on commodity hardware is not a way to increase the supported data rate. The bottleneck is between the NIC and the CPU, so there is nothing the GPU can do.
- On all specialized hardware-accelerated platforms, be it NPUs or FPGAs, the ability to support the maximum data rate of the interfaces is guaranteed by design. The result is visible in Section 4.5.2. But a slow processing applied to each received packet will still cause dropped packets.

Computation power

- CPUs provide high working frequencies and elaborated instruction sets, but a limited parallelism level. They are a good choice for light algorithms, or for algorithms that cannot be parallelized because of data dependencies. The DDoS detection algorithm in Section 3.2.2 is a good example of light non-parallel algorithm adapted to a CPU.
- GPUs can help CPUs because they are very good at massively parallel floating-point computations. They are not adapted to all applications because the communication between the CPU and the GPU is slow. If the number of similar computations to realize is not big enough, the time saved by the GPU will be wasted to send data to the GPU and to get the results. For traffic monitoring, GPU is not adapted

to accelerate a computation made for each packet. But it is useful if computations on a large number of packets can be aggregated before being sent to the GPU.

- NPUs are usually more basic than powerful CPUs, but they offer a higher level of parallelism. They also provide hardware-accelerated functions specialized for traffic monitoring. They are interesting for applications that rely heavily on these functions.
- FPGAs offer massive parallelism and a very low-level control, but they work at frequencies much lower than CPUs. They are interesting for algorithms that can be parallelized like the SVM algorithm in Section 4.3. They have difficulties with floating-point computations, so the algorithm should be converted to fixed-point if possible.

Flexibility

- The most flexible development platform is without a doubt the CPU with a pure software implementation. The development of a GUI is very easy, as well as the integration of variable configuration parameters in the algorithms. This is outlined by the BlockMon framework and its extreme flexibility 3.3.
- FPGAs are efficient when they use custom hardware to realize complex functions. Modifying the behaviour of the function usually requires a change of the custom hardware, which can only be done by reconfiguring the FPGA, a slow and complex process. This is what makes the implementation of the traffic classifier in Section 4.3 difficult to adapt to new situations.
- But flexibility can be obtained on any platform if it is a requirement considered during the design process. It usually implies a control of the hardware accelerator by a computer. This way a GUI can be used for configuration. This is the method used for the traffic generator in Section 5.3.

Ease of use

- The ease of use is a subjective concept. It depends on the knowledge of the user. An experienced C programmer will prefer software development, but a developer more used to FPGAs might be faster using an FPGA than a CPU.
- A lot of programmers are used to software development, and it is for sure the easiest platform to start with. But developing an application that supports high data rates actually requires a lot of expertise about the way a computer works. Getting the application to actually work on a machine is even more complicated, with a lot of parameters to tune to get good results. This is what I discovered when getting results about the performance of BlockMon in Section 3.5.2.

- FPGA development is very specific. It requires a methodology that has to be learnt before starting. For an experienced programmer, development can be fast. The longest task on FPGA usually is debugging, because the visibility on what happens inside the FPGA is limited. Debugging tools help the process, but it remains slow. It is particularly true when discovering a new board. Less debugging time was required for the traffic generator than for the traffic classifier because I already knew the board.

I have tested FPGAs and not GPUs or NPUs, although all platforms have different advantages. The reason is that FPGAs are the most low-level platform. It is very interesting for research, because everything that can be done on a CPU, a GPU or an NPU can be done on an FPGA. It is actually totally possible to implement a CPU, a GPU or an NPU on an FPGA. Of course, it is not always the best solution to get a working solution as fast as possible, but it is great to explore different possible architectures, with as few constraints as possible. Good solutions can then be ported to more efficient platforms.

6.3 Perspectives

Use cases for traffic monitoring seem to become more and more numerous. Governments want more control over the Internet, they want to intercept communications and to block access to some contents. Network operators are looking for new ways to make money, while unlimited broadband accesses get always cheaper. Offering new services with a guaranteed QoS is a potential solution. All of this requires a precise knowledge of the traffic flowing through the links of the network.

Although many people would like to keep the Internet free of any control, the importance the network has taken in everyone's life and in the global economy is so important, that some control is required. Without invading the privacy of their customers, Internet providers have to know what is transiting on their network, if only to keep the network up. Large scale attacks have the ability to take down big parts of the Internet. And as the Internet is more and more used for commerce and for communication, laws must apply on the Internet in the same way as they apply everywhere else. This means that the police needs some ways to act on exchanges happening online.

On the same time, the global quantity of data exchanged in networks keeps increasing, which means that the data rates increase too. So the need for high-speed traffic monitoring applications will only increase in the future.

The BlockMon framework could be used by operators to manage a flexible and powerful network of probes to get real-time data about their network. The goal of the DEMONS project was even to make operators collaborate to get a more global vision of the network, and to be able to react to attacks in a coordinated way. Although the TCP SYN flooding detection application I built on top of BlockMon is just a test application, it could be used as a base to develop an efficient network-wide DDoS detection system.

The high speed real-time traffic classification application I have developed could be very useful to apply some QoS rules to packets depending on the type of application that generated them. Future works on a way to automate the SVM learning phase, to facilitate the deployment of the application on new networks, would be needed.

The flexible open-source hardware-accelerated 20 Gb/s traffic generator is very promising for researchers who want to test their new algorithms using an affordable tool. Especially once it will be adapted to work on the widely available NetFPGA 10G board, which I hope to do soon. Anyone interested is welcome to use the generator and contribute to add new features.

To go even further in high data rates support, new development platforms are arriving on the market. Like for example a board with an FPGA and a 100 Gb/s interface. CPUs, GPUs and NICs evolve fast too. Publicly available NPUs seem to evolve less fast than other technologies. But manufacturers use custom-made NPUs that seem very powerful. An interesting point is that a custom NPU can always be implemented on an FPGA.

We saw that solutions mixing pure software development with hardware-accelerated functions on FPGA can be very efficient to provide both the performance improvements of FPGAs and the flexibility of software. This is the way our traffic generator works, and it can also be used in BlockMon with low-level blocks on FPGA. The architecture is always the same: the part in direct contact with the network is implemented on FPGA, aggregated data is exchanged between the computer and the FPGA, and the most high-level processing is done on the FPGA. But some new boards could make this kind of architectures even more efficient: Zynq boards from Xilinx [Xil14] include an ARM CPU directly connected to an FPGA. So mixing pure software and hardware-accelerated functions on the same board is possible. The communication between the CPU and the FPGA is more efficient on the board than using a PCIe bus, offering the possibility to get even better performance. And if the FPGA on the Zynq is not enough, it could communicate with a second FPGA on the same board. The latest Virtex 7 FPGAs provide much more computation power than the Virtex 5 FPGAs used in the NetFPGA 10G and Combo 20G boards. We saw with the traffic generator that a Virtex 5 FPGA provided an acceleration of a factor superior to 200 over a pure software implementation for SVM computation. Using a Virtex 7, the results would be even better. The architecture of the Zynq, with a very efficient communication between an FPGA and a CPU, could bring an improvement of a factor of at least 10 on the supported data rate, reaching at least 100 Gb/s.

We could wonder if hardware-accelerated solutions will become useless because of the rapid improvements in standard computer architectures. But CPUs are made to be generic. They can be used to play games, browse the Internet, make mathematical computations or process traffic. This genericity forces them to make trade-offs. This is why specialized hardware will remain more efficient, and as the needs will keep increasing, they will remain a solution to study.

Glossary

ADSL Asymmetric Digital Subscriber Line (ADSL) is a data transmission technology very widely used over copper telephone lines to provide fast Internet access. 78

API An Application Programming Interface (API) is a specified interface a program makes available, so as to communicate with other programs. It can be made of function calls, network requests.... 54

ASIC An Application-Specific Integrated Circuit (ASIC) is designed at hardware level using basic logic and arithmetic gates to realize a specific function at very high speed. 15, 46, 49

BGP Border Gateway Protocol (BGP) is used by network providers to communicate routes on the network. 26

botnet A botnet is a set of remotely-controlled machines used to perform together the same task. It is a way to manage efficiently tasks that require a lot of resources. 57

C4.5 C4.5 is a supervised learning algorithm used for classification. It is based on binary decision trees.. 89–92, 96, 123, 158

CMS A Count Min Sketch (CMS) is a probabilistic algorithm to store a list of counters in a constrained memory space. 10, 11, 16, 17, 20, 63, 64, 66, 71–75, 77, 78, 80–84, 104, 123, 157, 159

CPU A Central Processing Unit (CPU) is the integrated circuit used to make all basic operations in a computer. It may contain multiple cores to be able to process multiple operations concurrently. There may also be more than one CPU cooperating in a computer to increase the parallelism. 32, 35, 37–41, 43, 45, 48–50, 77, 79–83, 90, 129, 132, 134, 156–158, 161–164

CRC The Cyclic Redundancy Check (CRC) is a code used to detect errors in a transmitted message. Message data is used to compute a checksum appended to the message before transmission, and the same algorithm is used after transmission to check the received value. 145, 146

CUSUM The CUmulative SUM control chart (CUSUM) is an algorithm made to detect sudden changes in series of values. 10, 11, 16, 20, 60, 63–66, 71–74, 78, 82–84, 157

DDoS Distributed Denial of Service (DDoS) attacks use multiple computers connected to the Internet to send more traffic to a target than it can handle, so as to make it unresponsive. 15–17, 53–55, 57–62, 64–66, 72, 78–82, 84, 85, 125, 130, 131, 145, 157–159, 161, 163

DEMONS DEcentralized, cooperative, and privacy-preserving MONitoring for trustworthinesS (DEMONS) is the subject of the FP7 European project we contributed to. 10, 15, 29, 53, 55, 60–62, 67, 83, 84, 157, 163

DNS Domain Name Service (DNS) is a protocol that maintains an association between easy-to-remember domain names and routable IP addresses. It provides simpler addresses to contact machines. 26, 56, 57

DPI Deep Packet Inspection (DPI) is a traffic classification technique which consists in reading the full content of each received packet and check if it fits some pre-defined signatures. Each signature belongs to a class of applications to which the packet is then assigned. 32, 39, 86, 88, 89, 95

DSP Digital Signal Processing. 44, 46, 90, 91

FCS Frame Check Sequence (FCS) is the last field of an Ethernet frame. It is a kind of checksum computed from the whole frame data. It is used to check data integrity. 133, 137, 144

FIFO First In, First Out (FIFO) is a method used to queue data. All received data items are stored in an ordered way. The read item is always the one that was stored first. Once an item is read, it is removed from the queue. 141, 142, 147, 152

FPGA A Field-Programmable Gate Array (FPGA) is an integrated circuit that can be configured as many times as necessary at a very low level by connecting logical gates and registers together. The main languages used to represent the configuration are VHDL and Verilog. 9–11, 14–20, 23, 24, 30–32, 34, 40, 43–49, 51, 82–84, 89–91, 103, 108, 123–125, 128, 132–134, 138, 139, 143–145, 148, 149, 152, 153, 156, 158–164

FSM A Finite-State Machine (FSM) is an automaton used to control a process. It transitions from state to state depending on input data, and it outputs control signal depending on its internal state and input data.. 147

FTP File Transfer Protocol (FTP) is designed to transfer files between a client and a server. 86

GPU A Graphics Processing Unit (GPU) is a specialized integrated circuit designed for images manipulations. They are also used in other situations. They are particularly suited for highly parallel floating-point calculations on important amounts of data. 23, 24, 32, 34, 35, 39, 40, 43, 47, 48, 50, 82, 90, 156, 161–164

GUI A Graphical User Interface (GUI) is a communication tool between a computer and a human based on visual representations on a screen. It is the most current kind of interface used on computers.. 10, 11, 30, 67, 68, 71, 78, 83, 128, 130, 132, 134–136, 138, 142, 145, 146, 149, 152, 153, 157, 158, 160, 162

HTTP HyperText Transfer Protocol (HTTP) is the communication protocol used by the web. 58, 86, 87, 127

HTTPS HyperText Transfer Protocol Secure (HTTPS) is the use of the HTTP protocol over a security layer that provides website authentication and communication encryption. 86, 127

IANA The Internet Assigned Numbers Authority (IANA) controls the assignment of reserved UDP and TCP ports, among other Internet resources. 86, 87

ICMP Internet Control Message Protocol (ICMP) is used by network devices to request and send status messages. It is mostly known for its "ping" feature that is made to check the responsiveness of an IP address. 56, 133

IDMEF Intrusion Detection Message Exchange Format (IDMEF) is a message format for detailed network alert messages. 70, 73, 76, 77, 82, 157

IP Intellectual Property is a term used to design closed-source entities provided by third-parties in electronics. 48, 51, 152

IP Internet Protocol (IP) is the base protocol used on the Internet.. 26, 28, 29, 40, 56, 58, 61–64, 66, 69, 70, 73–76, 80, 86, 88, 89, 91, 92, 100, 106, 127, 131, 133, 143, 144, 146, 157

IPFIX Internet Protocol Flow Information eXport (IPFIX) is a protocol made to communicate network-specific information over the network. 70, 71, 76

IPv4 Internet Protocol version 4 (IPv4) is the current version of the protocol. It is slowly being replaced by version 6.. 28, 63, 100, 143–145, 148

IPv6 Internet Protocol version 6 (IPv6) is a new version of the protocol being currently deployed. It has not yet replaced the current version 4, but should soon.. 39, 100, 144, 148

IRC Internet Relay Chat (IRC) is a text-based instant communication protocol. 57, 86

JSON JavaScript Object Notation (JSON) is a lightweight text-based data exchange format. 135

LDPC A Low-Density Parity-Check (LDPC) code is a method to add information to a message, so as to recover it after transmission if errors have been inserted. 108

LUT A Look-Up Table (LUT) realizes a function that takes a word made of a certain number of bits as input, and outputs another word made of another number of bits, depending only on the input. It can be configured to link each input word to any output word. 43, 44, 46

MD5 Message Digest 5 (MD5) is a cryptographic hash function commonly used to check data integrity. 107, 108

NAPI New Application Programming Interface (NAPI) is an improvement of the Linux kernel to support high-speed networking with a lighter load on the CPU. 36

NIC A Network Interface Card (NIC) is a board that can be connected to a computer to provide it a network connection. The most common wired boards use the low-level Ethernet protocol. They usually manage the lowest parts of the network stack to help the CPU. A driver is needed for the CPU to communicate with the board. 23, 35–40, 50, 69, 70, 77, 79–82, 84, 128, 129, 156–158, 161, 164

NPU A Network Processing Unit (NPU) is a specialized integrated circuit designed for network applications. It has a direct access to network interfaces, and some specialized instructions (CRC computation for example) make frequent operations on packets faster. 23, 24, 32, 40–43, 48, 49, 51, 82, 156, 161–164

NUMA Non Uniform Memory Access (NUMA) is a computer architecture where each processor is assigned an area of RAM with low access delays. Other areas are accessible but slower.. 38, 79

OSCAR Overlay networks Security: Characterization, Analysis and Recovery (OSCAR) is the name of a French Research project in which Télécom Bretagne was implied. 53, 78

P2P Peer-to-peer (P2P) is a kind of communication system where each implied computer is both a server and a client. It is often used for file sharing.. 57, 86, 87, 91

pcap pcap (Packet CAPture) is both an API to capture packets on a network interface, and a file format used to save a trace of captured packets. Implementations exist for Windows and Linux. 132

PCIe Peripheral Component Interconnect express (PCIe) is a standard for a local serial communication bus. It is used to connect extension cards to a motherboard. 39, 45, 79, 80, 132, 134, 164

QoS Quality of Service (QoS) is a measure of the quality assured on a netwokr link. Many parameters can be used: inter-packet delay, jitter, data rate, etc. 33, 85, 86, 103, 163, 164

RAM A Random Access Memory (RAM) is a memory with fast read and write operations at any address. The memory is volatile, which means that data is lost when power is cut. 38, 79, 82, 102, 103, 106

RPC Remote Procedure Call (RPC) is a network protocol designed to allow different programs to make transparent procedure calls to one another. 67

RSS Receive Side Scaling (RSS) is a technology used by some NICs to send received packets to the host CPU in different queues, which can be handled in parallel by the CPU.. 16, 35, 36, 70, 79, 81, 83

SFP+ An enhanced Small Form-factor Pluggable (SFP+) is a small pluggable transceiver that converts optical signal to electrical signal and vice versa. 46

SHA Secure Hash Algorithm (SHA) is a family of cryptographic hash functions. 107, 108

SMS Short Message Service (SMS) is a protocol designed to send short text messages to mobile phones. 57

SSH Secure SHell (SSH) is a secure remote control and file transfer protocol. 58, 86

SVM Support Vector Machine (SVM) is a supervised learning algorithm used for classification. It is based on finding hyperplanes between categories.. 10, 11, 17, 18, 20, 85, 89–99, 101, 107–111, 115–124, 158, 159, 162, 164

TCP Transmission Control Protocol (TCP) is a transport protocol very commonly used over IP. It is designed for reliable connected data transfer. 10, 26, 56–58, 60–66, 70–74, 76, 78, 80–82, 84, 86–89, 91, 126, 131, 143–146, 148, 157, 163

UDP User Datagram Protocol (UDP) is a transport protocol very commonly used over IP. It is designed for simple data transfer. 26, 56, 60, 86–89, 91, 131, 143, 145, 146, 148

VoIP Voice over IP (VoIP) is a category of applications that allow to make calls over the Internet. 55, 61, 85

XML eXtensible Markup Language (XML) is a generic text-based description language. 67, 68, 71, 76

Bibliography

[ABR03] D. Anguita, A. Boni, and S. Ridella. A digital architecture for support vector machines: theory, algorithm, and FPGA implementation. *Neural Networks, IEEE Transactions on*, 14(5):993–1009, 2003.

[ADPF+08] G. Antichi, A. Di Pietro, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Design of a high performance traffic generator on network processor. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 438–441, 2008.

[AFK+12] Rafael Antonello, Stenio Fernandes, Carlos Kamienski, Djamel Sadok, Judith Kelner, IstváN GóDor, GéZa Szabó, and Tord Westholm. Deep Packet Inspection tools and techniques in commodity platforms: Challenges and trends. *J. Netw. Comput. Appl.*, 35(6):1863–1878, November 2012.

[AIN13] Yuki Ago, Yasuaki Ito, and Koji Nakano. An efficient implementation of a Support Vector Machine in the FPGA. *Bulletin of Networking, Computing, Systems, and Software*, 2(1), 2013.

[AL11] Alcatel-Lucent. FP3: 400G network processor. http://www3.alcatel-lucent.com/products/fp3/, 2011. [Online].

[Ama13] Amazon. Amazon.com: nvidia tesla. http://www.amazon.com/s/?field-keywords=nvidia%20tesla, 2013. [Online].

[AMY09] S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131, 2009.

[And98] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, pages 191–200, New York, NY, USA, 1998. ACM.

[AOP+08] Carsten Albrecht, Christoph Osterloh, Thilo Pionteck, Roman Koch, and Erik Maehle. An application-oriented synthetic net-

work traffic generator. In *22nd European Conference on Modelling and Simulation.*, 2008.

[APRS06] Davide Anguita, Stefano Pischiutta, Sandro Ridella, and Dario Sterpi. Feed-Forward Support Vector Machine Without Multipliers. *IEEE Transactions on Neural Networks*, 17(5):1328–1331, 2006.

[AR12] Mohammed Alenezi and Martin Reed. Methodologies for detecting DoS/DDoS attacks against network servers. In *ICSNC 2012, The Seventh International Conference on Systems and Networks Communications*, pages 92–98, 2012.

[ASGM13] Gianni Antichi, Muhammad Shahbaz, Stefano Giordano, and Andrew Moore. From 1g to 10g: code reuse in action. In *Proceedings of the first edition workshop on High performance and programmable networking*, HPPN '13, pages 31–38, 2013.

[ATT13] ATT. Global IP network latency. http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html, 2013. [Online; accessed 21-March-2013].

[BAAZ10] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, January 2010.

[BBCR06] Raffaele Bolla, Roberto Bruschi, Marco Canini, and Matteo Repetto. A high performance IP traffic generation tool based on the intel IXP2400 network processor. In *Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements*, pages 127–142. Springer, 2006.

[BDP10] A. Botta, A. Dainotti, and A. Pescapé. Do you trust your software-based traffic generator? *Communications Magazine, IEEE*, 48(9):158–165, sept. 2010.

[BDP12] Alessio Botta, Alberto Dainotti, and Antonio Pescapé. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531 – 3547, 2012.

[BDPGP12a] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. Flexible high performance traffic generation on commodity multi–core platforms. *Traffic Monitoring and Analysis*, pages 157–170, 2012.

[BDPGP12b] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. On multi-gigabit packet capturing with multi-core commodity hardware. In *Proceedings of the 13th international*

conference on Passive and Active Measurement, PAM'12, pages 64–73, 2012.

[BEM+10] Michaela Blott, Jonathan Ellithorpe, Nick McKeown, Kees Vissers, and Hongyi Zeng. FPGA research design platform fuels network advances. *Xilinx Xcell Journal*, (73), 2010.

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152, 1992.

[BMM+11] Paola Bermolen, Marco Mellia, Michela Meo, Dario Rossi, and Silvio Valenti. Abacus: Accurate behavioral classification of p2p-tv traffic. *Computer Networks*, 55(6):1394 – 1411, 2011.

[BR09] Mario Baldi and Fulvio Risso. Towards effective portability of packet handling applications across heterogeneous hardware platforms. In *Active and Programmable Networks*, pages 28–37. Springer, 2009.

[BRP12] T. Bujlow, T. Riaz, and J.M. Pedersen. Classification of http traffic based on c5.0 machine learning algorithm. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 882–887, 2012.

[BTS06] Laurent Bernaille, Renata Teixeira, and Kave Salamatian. Early application identification. In *Proceedings of the 2006 ACM CoNEXT conference*, CoNEXT '06, pages 6:1–6:12, 2006.

[Car09] AUSTIN Carpenter. CUSVM: A CUDA implementation of support vector classification and regression. 2009.

[CCR11] Niccolò Cascarano, Luigi Ciminiera, and Fulvio Risso. Optimizing Deep Packet Inspection for high-speed traffic analysis. *J. Netw. Syst. Manage.*, 19(1):7–31, March 2011.

[CDGS07] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Computer Communication Review*, 37(1):5–16, 2007.

[Cis13] Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2012–2017, 2013.

[CJM05] Evan Cooke, Farnam Jahanian, and Danny McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of the USENIX SRUTI Workshop*, volume 39, page 44, 2005.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[Cle] Clear Foundation. l7-filter: application layer packet classifier for Linux. http://l7-filter.clearfoundation.com/.

[CLS+08] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107, 2008.

[CM05] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[Cow03] C. Cowan. Software security for open-source systems. *Security Privacy, IEEE*, 1(1):38–45, 2003.

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (json), 2006. RFC 4627.

[CSK08] Bryan Christopher Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Technical Report No. UCB/EECS-2008-11*. EECS Department, University of California, Berkeley, 2008.

[CT13] B. Claise and B. Trammell. Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information, 9 2013. RFC 7011.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[DCF07] H. Debar, D. Curry, and B. Feinstein. The intrusion detection message exchange format (IDMEF), 3 2007. RFC 4765.

[DdDPSR08] A. Dainotti, W. de Donato, A. Pescape, and P. Salvo Rossi. Classification of network traffic via packet-level hidden markov models. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5, 2008.

[DEM13] DEMONS. BlockMon source code. https://github.com/blockmon/blockmon, 2013.

[DHV01] J. Deepakumara, H.M. Heys, and R. Venkatesan. FPGA implementation of MD5 hash algorithm. In *Electrical and Computer Engineering, 2001. Canadian Conference on*, volume 2, pages 919–924, 2001.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[DPC12] A. Dainotti, A. Pescape, and K.C. Claffy. Issues and future directions in traffic classification. *Network, IEEE*, 26(1):35–40, 2012.

[dPHB+13] Andrea di Pietro, Felipe Huici, Nicola Bonelli, Brian Trammell, Petr Kastovsky, Tristan Groleat, Sandrine Vaton, and Maurizio Dusi. Toward composable network traffic measurement. In *INFOCOM 2013: 32nd IEEE Conference on Computer Communications*, 2013.

[dRR13] Pedro Maria Santiago del Rio and Javier Aracil Rico. *Internet Traffic Classification for High-Performance and Off-The-Shelf Systems*. PhD thesis, Universidad autonoma de Madrid, 2013.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012.

[EG11] A. Este and F. Gringoli. On-line SVM traffic classification. In *Proceedings of the 7th IWCMC Conference (IWCMC TRAC'2011)*, 2011.

[EGS09] A. Este, F. Gringoli, and L. Salgarelli. Support vector machines for TCP traffic classification. *Computer Networks*, 53(14):2476 – 2490, 2009.

[FGH10] B. Fontaine, T. Groléat, and F. Hubert. Surveillance réseau sur NetFPGA. http://trac.benoute.fr/netfpga/attachment/wiki/Livrables/rapport_technique.pdf, 2010.

[FHD+09] F. Fusco, F. Huici, L. Deri, S. Niccolini, and T. Ewald. Enabling high-speed and extensible real-time communications monitoring. In *Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on*, pages 343–350, 2009.

[FML+03] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the sprint IP backbone. *Network, IEEE*, 17(6):6–16, 2003.

[FSdLFGLJ+13] Alysson Feitoza Santos, Stenio Flavio de Lacerda Fernandes, Petrônio Gomes Lopes Júnior, Djamel Fawzi Hadj Sadok, and Geza Szabo. Multi-gigabit traffic identification on GPU. In *Proceedings of the first edition workshop on High performance and programmable networking*, pages 39–44, 2013.

[GHY+13] Lu Gang, Zhang Hongli, Zhang Yu, M.T. Qassrawi, Yu Xiangzhan, and Peng Lizhi. Automatically mining application signatures for lightweight Deep Packet Inspection. *Communications, China*, 10(6):86–99, 2013.

[GNES12] F. Gringoli, L. Nava, A. Este, and L. Salgarelli. MTCLASS: enabling statistical traffic classification of multi-gigabit aggregates on inexpensive hardware. In *Proceedings of the 8th IWCMC Conference (IWCMC TRAC'2012)*, 2012.

[GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, FPGA '04, pages 162–170, New York, NY, USA, 2004. ACM.

[GP09] G.Gomez and P.Belzarena. Early Traffic Classification using Support Vector Machines. In *Fifth International Latin American Networking Conference (LANC'09)*, 2009.

[Gro13] T. Groléat. Open-source hardware traffic generator. https://github.com/tristan-TB/hardware-traffic-generator, 2013. [Online].

[GSD+09a] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. Claffy. Gt: picking up the truth from the ground for internet traffic. *ACM SIGCOMM CCR*, 2009.

[GSD+09b] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K.C. Claffy. GT: picking up the truth from the ground for Internet traffic. *ACM SIGCOMM Computer Communication Review*, 39(5):13–18, 2009.

[GSG+12] M. Ghobadi, G. Salmon, Y. Ganjali, M. Labrecque, and J.G. Steffan. Caliper: Precise and responsive traffic generator. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 25–32, aug. 2012.

[Han13] Troy D. Hanson. Ut hash. http://troydhanson.github.com/uthash/, 2013. [Online; accessed 1-March-2013].

[Hep03] Andrew Heppel. An introduction to network processors. *Roke Manor Research Ltd., White paper*, 2003.

[HJPM10] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010.

[HSL09] Nan Hua, Haoyu Song, and T. V. Lakshman. Variable-stride multi-pattern matching for scalable Deep Packet Inspection. In *INFOCOM 2009, IEEE*, pages 415–423, 2009.

[Ian06] Gianluca Iannaccone. Fast prototyping of network data mining applications. In *Passive and Active Measurement Conference*, 2006.

[IDNG08] K.M. Irick, M. DeBole, V. Narayanan, and A. Gayasen. A hardware efficient Support Vector Machine architecture for FPGA. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 304–305, 2008.

[Int07] Intel. Ixp4xx product line of network processors. http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html, 2007. [Online].

[Int13] Intel. Intel ethernet server adapter i350 product family. http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/ethernet-server-adapter-i350.html, 2013. [Online].

[IP11] Sunghwan Ihm and Vivek S. Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 295–312, 2011.

[IT13] Invea-Tech. COMBO-20G FPGA Card. https://www.invea-tech.com/products-and-services/fpga-cards/combo-20g, May 2013. [Online].

[ixi12] 10G Ethernet test solution. http://www.ixiacom.com/products/interfaces/display?skey=in_10g_universal, 2012. [Online; accessed 6-February-2013].

[JG10] Weirong Jiang and Maya Gokhale. Real-Time Classification of Multimedia Traffic Using FPGA. In *Field-Programmable Logic and Applications*, pages 56–63, 2010.

[JP12] Shuchi Juyal and Radhika Prabhakar. A comprehensive study of ddos attacks and defense mechanisms. *Journal of Information and Operations Management*, 3(1), 2012.

[JR86] R. Jain and S. Routhier. Packet trains–measurements and a new model for computer network traffic. *Selected Areas in Communications, IEEE Journal on*, 4(6):986–995, 1986.

[JY04] Shuyuan Jin and D.S. Yeung. A covariance analysis model for ddos attack detection. In *Communications, 2004 IEEE International Conference on*, volume 4, pages 1882–1886 Vol.4, 2004.

[KCF+08] H. Kim, KC Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and KY Lee. Internet traffic classification demystified: myths, caveats, and the best practices. *Proc. of ACM CoNEXT*, 2008.

[KKZ+11] P. Korcek, V. Kosar, M. Zadnik, K. Koranda, and P. Kastovsky. Hacking NetCOPE to run on NetFPGA-10G. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 217–218, oct. 2011.

[KMC+00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[KPF05] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 229–240. ACM, 2005.

[LHCK04] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *ACM Trans. Embed. Comput. Syst.*, 3(3):614–633, August 2004.

[LKJ+10] Yeon-sup Lim, Hyun-chul Kim, Jiwoong Jeong, Chong-kwon Kim, Ted "Taekyoung" Kwon, and Yanghee Choi. Internet traffic classification demystified: on the sources of the discriminative power. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 9:1–9:12, 2010.

[LKL12] Wangbong Lee, Dong Won Kang, and Joon Kyung Lee. A plug-in node architecture for dynamic traffic control. In *EMERGING 2012, The Fourth International Conference on Emerging Network Intelligence*, pages 19–21, 2012.

[LMW+07] J.W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. 2007.

[LSS+09] Martin Labrecque, J Gregory Steffan, Geoffrey Salmon, Monia Ghobadi, and Yashar Ganjali. NetThreads: Programming NetFPGA with threaded software. In *NetFPGA Developers Workshop '09*, 2009.

[LTWW94] W.E. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson. On the self-similar nature of ethernet traffic (extended version). *Networking, IEEE/ACM Transactions on*, 2(1), 1994.

[LZB11] Guangdeng Liao, Xia Zhu, and L. Bnuyan. A new server i/o architecture for high speed networks. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 255–265, 2011.

[LZL+09] Ke Li, Wanlei Zhou, Ping Li, Jing Hai, and Jianwen Liu. Distinguishing ddos attacks from flash crowds using probability metrics. In *Network and System Security, 2009. NSS '09. Third International Conference on*, pages 9–17, 2009.

[MA12] Milton L. Mueller and Hadi Asghari. Deep Packet Inspection and bandwidth management: Battles over BitTorrent in Canada and the United States. *Telecommunications Policy*, 36(6):462 – 475, 2012.

[Mar08] Zadnik Martin. NetFlow probe on NetFPGA. http://www.liberouter.org/~xzadni00/netflowprobedoc.pdf, December 2008. [Online; accessed 1-October-2013].

[Mar13] Marvell. Network processors. http://www.marvell.com/network-processors/, 2013. [Online].

[Mat13] Prince Matthew. The DDoS that knocked spamhaus offline (and how we mitigated it). http://blog.cloudflare.com/the-ddos-that-knocked-spamhaus-offline-and-ho, March 2013. [Online; accessed 15-October-2013].

[MBdC+06] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 199–208, 2006.

[MCMM06] Robert P. McEvoy, F.M. Crowe, C.C. Murphy, and William P. Marnane. Optimisation of the SHA-2 family of hash functions on FPGAs. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, 2006.

[MdRR+12] V. Moreno, P.M.S. del Rio, J. Ramos, J.J. Garnica, and J.L. Garcia-Dorado. Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines. *Communications Letters, IEEE*, 16(11):1888–1891, 2012.

[MK08] T. Martmek and M. Kosek. Netcope: Platform for rapid development of network applications. In *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, pages 1–6. IEEE, 2008.

[MKK+01] David Moore, Ken Keys, Ryan Koga, Edouard Lagache, and K. C. Claffy. The coralreef software suite as a tool for system and network administrators. In *Proceedings of the 15th USENIX conference on System administration*, LISA '01, pages 133–144, 2001.

[MMSH01] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. NetBench: a benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 39–42, Piscataway, NJ, USA, 2001. IEEE Press.

[MP05] AndrewW. Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *Passive and Active Network Measurement*, volume 3431 of *Lecture Notes in Computer Science*, pages 41–54. 2005.

[MSD+07] Jun Mu, S. Sezer, Gareth Douglas, D. Burns, E. Garcia, M. Hutton, and K. Cackovic. Accelerating pattern matching for dpi. In *SOC Conference, 2007 IEEE International*, pages 83–86, 2007.

[Mut04] S. Muthukrishnan. MassDAL Code Bank – Sketches, Frequent Items, Changes. http://www.cs.rutgers.edu/~muthu/massdal-code-index.html, 2004.

[NA08] T.T.T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys Tutorials, IEEE*, 10(4):56–76, 2008.

[Net12] NetFPGA. NetFPGA 10G. http://netfpga.org/10G_specs.html, 2012. [Online].

[New13] BBC News. Thirteen plead guilty to anonymous hack of paypal site. http://www.bbc.co.uk/news/business-25327175, December 2013. [Online; accessed 11-December-2013].

[NI10] Alastair Nottingham and Barry Irwin. Parallel packet classification using GPU co-processors. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, pages 231–241, New York, NY, USA, 2010. ACM.

[NP12] Anh Nguyen and Lei Pan. Detecting sms-based control commands in a botnet from infected android devices. In *ATIS 2012: Proceedings of the 3rd Applications and Technologies in Information Security Workshop*, pages 23–27. School of Information Systems, Deakin University, 2012.

[OGI+12] A.-C. Orgerie, P. Goncalves, M. Imbert, J. Ridoux, and D. Veitch. Survey of network metrology platforms. In *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on*, pages 220–225, 2012.

[PB10] M. Papadonikolakis and C. Bouganis. A novel FPGA-based SVM classifier. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 283–286, 2010.

[Pus12] Viktor Pus. Hardware acceleration for measurements in 100 gb/s networks. In *Dependable Networks and Services*, volume 7279 of *Lecture Notes in Computer Science*, pages 46–49. Springer Berlin Heidelberg, 2012.

[PYR+13] PyungKoo Park, SeongMin Yoo, HoYong Ryu, Cheol Hong Kim, Su il Choi, Jaehyung Park, and JaeCheol Ryou. Service-oriented ddos detection mechanism using pseudo state in a flow router. In *Information Science and Applications (ICISA), 2013 International Conference on*, pages 1–4, 2013.

[QXH+07] Yaxuan Qi, Bo Xu, Fei He, Baohua Yang, Jianming Yu, and Jun Li. Towards high-performance flow-level packet processing on multi-core network processors. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 17–26, 2007.

[RAMV07] J.J. Rodriguez-Andina, M.J. Moure, and M.D. Valdes. Features, design tools, and application domains of FPGAs. *Industrial Electronics, IEEE Transactions on*, 54(4):1810–1823, 2007.

[RDC12] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 Gbit/s line rate packet processing using commodity hardware: Survey and new proposals. http://luca.ntop.org/10g.pdf, 2012. [Online].

[Riz12] Luigi Rizzo. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[RLAM12] M.Muzaffar Rao, Kashif Latif, Arshad Aziz, and Athar Mahboob. Efficient FPGA implementation of secure hash algorithm grøstl – SHA-3 finalist. In BhawaniShankar Chowdhry, FaisalKarim Shaikh, DilMuhammadAkbar Hussain, and MuhammadAslam Uqaili, editors, *Emerging Trends and Applications in Information Communication Technologies*, volume 281 of *Communications in Computer and Information Science*, pages 361–372. Springer Berlin Heidelberg, 2012.

[Rya13] Cox Ryan. 5 notorious DDoS attacks in 2013 : Big problem for the internet of things. http://siliconangle.com/blog/2013/08/26/5-notorious-ddos-attacks-in-2013-big-problem-for-the-internet-of-things/, August 2013. [Online; accessed 15-October-2013].

[SAG+13] Muhammad Shahbaz, Gianni Antichi, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick Feamster, Nick McKeown, Bob Felderman, Michaela Blott, Andrew Moore, and Philippe Owezarski. Architecture for an open source network tester. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 123–124, 2013.

[Sch06] Andrew Schmitt. The future of network processing units doesn't look too rosy. 2006.

[SdRRG+12] Pedro M. Santiago del Rio, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 ACM conference on Internet Measurement Conference*, pages 65–72, 2012.

[SDS+06] Vyas Sekar, Nick G Duffield, Oliver Spatscheck, Jacobus E van der Merwe, and Hui Zhang. Lads: Large-scale automated ddos detection system. In *USENIX Annual Technical Conference, General Track*, pages 171–184, 2006.

[SDTL05] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *SIGCOMM Comput. Commun. Rev.*, 35(4):181–192, August 2005.

[SEJK08] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM Comput. Commun. Rev.*, 38(4):207–218, August 2008.

[SGV+10] Géza Szabó, István Gódor, András Veres, Szabolcs Malomsoky, and Sándor Molnár. Traffic classification over Gbit speed with commodity hardware. *IEEE J. Communications Software and Systems*, 5, 2010.

[SH13] Vandana Singh and Lila Holt. Learning and best practices for learning in open-source software communities. *Computers & Education*, 63:98–108, 2013.

[SK12] Mark Scanlon and Tahar Kechadi. Peer-to-peer botnet investigation: A review. In *Future Information Technology, Application, and Service*, volume 179 of *Lecture Notes in Electrical Engineering*, pages 231–238. Springer Netherlands, 2012.

[SKB04] J. Sommers, H. Kim, and P. Barford. Harpoon: a flow-level traffic generator for router and network tests. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 392–392. ACM, 2004.

[SKKP12] Stavros N. Shiaeles, Vasilios Katos, Alexandros S. Karakos, and Basil K. Papadopoulos. Real time {DDoS} detection using fuzzy estimators. *Computers & Security*, 31(6):782 – 790, 2012.

[SKS+10] Kwangsik Shin, Jinhyuk Kim, Kangmin Sohn, Changjoon Park, and Sangbang Choi. Online gaming traffic generator for reproducing gamer behavior. In *Entertainment Computing - ICEC 2010*, volume 6243 of *Lecture Notes in Computer Science*, pages 160–170. 2010.

[SMV10] S. Stoev, G. Michailidis, and J. Vaughan. On global modeling of backbone network traffic. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5, 2010.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[SRB01] Shriram Sarvotham, Rudolf Riedi, and Richard Baraniuk. Connection-level analysis and modeling of network traffic. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 99–103, 2001.

[SSG11] M. Sanlı, E.G. Schmidt, and H.C. Güran. FPGEN: A fast, scalable and programmable traffic generator for the performance evaluation of high-speed computer networks. *Performance Evaluation*, 68(12):1276–1290, 2011.

[SVG10] O. Salem, S. Vaton, and A. Gravey. A scalable, efficient and informative approach for anomaly-based intrusion detection systems: theory and practice. *International Journal of Network Management*, 20(5):271–293, 2010.

[Tar05] A.G. Tartakovsky. Asymptotic performance of a multichart CUSUM test under false alarm probability constraint. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on*, pages 320–325, 2005.

[TDT11] A. Tockhorn, P. Danielis, and D. Timmermann. A configurable FPGA-based traffic generator for high-performance tests of packet processing systems. In *ICIMP 2011, The Sixth International Conference on Internet Monitoring and Protection*, pages 14–19, 2011.

[Tec13] EZChip Technologies. Products. http://www.ezchip.com/products.htm, 2013. [Online].

[TRBK06] A.G. Tartakovsky, B.L. Rozovskii, R.B. Blazek, and Hongjoong Kim. A novel approach to detection of intrusions in computer networks via adaptive sequential and batch-sequential change-point detection methods. *Signal Processing, IEEE Transactions on*, 54(9):3372–3382, 2006.

[TSMP13] Da Tong, Lu Sun, Kiran Matam, and Viktor Prasanna. High throughput and programmable online traffic classifier on FPGA. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '13, pages 255–264, 2013.

[ULBH08] Sain-Zee Ueng, Melvin Lathara, SaraS. Baghsorkhi, and WenmeiW. Hwu. Cuda-lite: Reducing GPU programming complexity. In *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2008.

[Und04] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, FPGA '04, pages 171–180, 2004.

[VM04] Harrick Vin and Jayaram Mudigonda. A programming environment for packet-processing systems: Design considerations. In *In Workshop on Network Processors & Applications - NP3*, pages 14–15, 2004.

[VPI11] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 297–308, 2011.

[VV09] K.V. Vishwanath and A. Vahdat. Swing: Realistic and responsive network traffic generation. *Networking, IEEE/ACM Transactions on*, 17(3):712–725, 2009.

[WA11] Pu Wang and Ian F. Akyildiz. Spatial correlation and mobility-aware traffic modeling for wireless sensor networks. *IEEE/ACM Trans. Netw.*, 19(6):1860–1873, December 2011.

[WCM09] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, 2009.

[WZA06] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM SIGCOMM Computer Communication Review*, 2006.

[xen12] XenaCompact. http://www.xenanetworks.com/html/xenacompact.html, 2012. [Online; accessed 6-February-2013].

[XIK+12] Guowu Xie, M. Iliofotou, R. Keralapura, Michalis Faloutsos, and A. Nucci. Subflow: Towards practical flow-level traffic classification. In *INFOCOM, 2012 Proceedings IEEE*, pages 2541–2545, 2012.

[Xil14] Xilinx. Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/, January 2014. [Online].

[XWZ13] Yibo Xue, Dawei Wang, and Luoshi Zhang. Traffic classification: Issues and challenges. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 545–549, 2013.

[YCM11] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. Progme: towards programmable network measurement. *IEEE/ACM Trans. Netw.*, 19(1):115–128, February 2011.

[YZD08] Shui Yu, Wanlei Zhou, and R. Doss. Information theory based detection against network behavior mimicking ddos attacks. *Communications Letters, IEEE*, 12(4):318–321, 2008.

[ZSGK09] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Characteristics of youtube network traffic at a campus network – measurements, models, and implications. *Computer Networks*, 53(4):501 – 514, 2009.

[ZXZW13] Jun Zhang, Yang Xiang, Wanlei Zhou, and Yu Wang. Unsupervised traffic classification using flow statistical properties and IP packet payload. *Journal of Computer and System Sciences*, 79(5):573 – 585, 2013.